Review

- Direct-mapped caches suffer from conflict misses
  - 2 memory blocks mapping to same block knock each other out as program bounces from 1 memory location to next
- One way to solve: set-associativity
  - Memory block maps into more than 1 cache block
  - N-way: n possible places in cache to hold a memory block
- N-way Cache of 2^N blocks: 2^N ways x 2^M sets
- Multi-level caches
  - Optimize first level to be fast!
  - Optimize 2nd and 3rd levels to minimize the memory access penalty

Recap: C Memory Management

- Program's address space contains 4 regions:
  - **stack**: local variables, grows downward
  - **heap**: space requested for pointers via malloc(), resize dynamically, grows upward
  - **static data**: variables declared outside main, does not grow or shrink
  - **code**: loaded when program starts, does not change

Recap: Where are Variables Allocated?

- If declared outside a procedure, allocated in "static" storage
- If declared inside procedure, allocated on the "stack" and freed when procedure returns
  ```c
  int myGlobal;
  main() {
    int myTemp;
  }
  ```

Recap: The Stack

- Stack frame includes:
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- Stack frames contiguous
  - blocks of memory; stack pointer indicates top of stack frame
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames
Recap: The Stack

- Last In, First Out (LIFO) data structure

```c
main ()
{
    a (0);
    void a (int n)
    {
        b (1);
        void b (int n)
        {
            c (2);
            void c (int n)
            {
                d (3);
            }
        }
    }
}
```

Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order

Managing the Heap

- C supports five functions for heap management:
  - `malloc()`, `calloc()`, `free()`, `cfree()`, `realloc()`
    - `malloc()`: Allocates a block of uninitialised memory
    - `calloc()`: Allocates a block of uninitialised memory and sets it to 0
    - `free()`: Deallocates a previously allocated memory block
    - `cfree()`: Deallocates a previously allocated memory block, but does not check the contents
    - `realloc()`:
    
- Example:
  ```c
  int *ip;
  ip = malloc(sizeof(int));
  struct treeNode *tp;
  tp = malloc(sizeof(struct treeNode));
  ```

Common Memory Problems

- Using uninitialized values
- Using memory that you don’t own
  - Deallocated stack or heap variable
  - Out-of-bounds reference to stack or heap array
- Using NULL or garbage data as a pointer
- Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
- Memory leaks (you allocated something you forgot to later free)

Managing the Heap

- `free()`: Releases memory allocated by `malloc()`
  - `ip` is a pointer containing the address originally returned by `malloc()`
  - `ip = malloc(sizeof(int));`...`free(ip);` /* Can you free(ip) after ip++? */
  ```c
  struct treeNode *tp;
  tp = malloc(sizeof(struct treeNode));
  free(ip);
  ```
  - When insufficient free memory, `malloc()` returns NULL pointer; Check for it!
  ```c
  if (ip = malloc(sizeof(int));) == NULL)
  {
      printf("Memory is FULL!
      exit(1);
  }
  ```
  - When you free memory, you must be sure that you pass the original address returned from `malloc()` to `free()`. Otherwise, system exception (or worse!)

Memory Debugging Tools

- Runtime analysis tools for finding memory errors
  - Dynamic analysis tool: collects information on memory management while program runs
  - Contrast with static analysis tool like lint, which analyzes source code without compiling or executing it
  - No tool is guaranteed to find ALL memory bugs – this is a very challenging programming language research problem
  - Runs 10X slower

Valgrind

[http://valgrind.org](http://valgrind.org)
Using Memory You Don’t Own

• What is wrong with this code?

```c
int *ipr, *ipw;
void ReadMem() {
    int i, j;
    *ipr = malloc(4 * sizeof(int));
    i = *(ipr - 1000); j = *(ipr + 1000);
    free(ipr);
}
void WriteMem() {
    *ipw = malloc(5 * sizeof(int));
    *(ipw - 1000) = 0; *(ipw + 1000) = 0;
    free(ipw);
}
```

How are Malloc/Free implemented?

• Underlying operating system allows malloc library to ask for large blocks of memory to use in heap (using Unix sbrk call)
• C Malloc library creates data structure inside unused portions to track free space

Simple Slow Malloc Implementation

Initial Empty Heap Space from Operating System

Free Space

Malloc library creates linked list of empty blocks (one block initially)

Object 1

Free

First allocation chews up space from start of free space

After many mallocs and frees, have potentially long linked list of odd-sized blocks
Frees link block back onto linked list — might merge with neighboring free space

Faster malloc implementations

• Keep separate pools of blocks for different sized objects
• “Buddy allocators” always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks:

Power-of-2 “Buddy Allocator”

Alloc Implementations

• All provide the same library interface, but can have radically different implementations
• Uses headers at start of allocated blocks and space in unallocated memory to hold malloc’s internal data structures
• Rely on programmer remembering to free with same pointer returned by alloc
• Rely on programmer not messing with internal data structures accidentally!
Faulty Heap Management
• 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();
}

Faulty Heap Management
• What is wrong with this code?
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  ... ...
  plk++;
}

Using Memory You Haven’t Allocated
• What is wrong with this code?
void StringManipulate() {
  const char *name = “Safety Critical”;
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = ‘\0’;
  printf(“%s\n”, str);
}

Using Memory You Don’t Own
• What’s 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; j<MAXSIZE-1 && j<strlen(s1); i++, j++) {
    result[i] = s1[j];
  }
  for (j=0; j<MAXSIZE-1 && j<strlen(s2); i++, j++) {
    result[j] = s2[j];
  }
  result[++i] = ‘\0’;
  return result;
}
Using Memory You Don’t Own

• What is wrong with this code?

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

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

Managing the Heap

• realloc(p, size):
  - Resize a previously allocated block at p to a new size
  - If p is NULL, then realloc behaves like malloc
  - If size is 0, then realloc behaves like free, deallocating the block from the heap
  - Returns new address of the memory block; NOTE: it is likely to have moved!

```c
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}
```

Using Memory You Don’t Own

• What is wrong with this code?

```c
int* fill_fibonacci(int *fib, int size) {
    int i;
    init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++) {
        fib[i] = fib[i-1] + fib[i-2];
    }
    return fib;
}
```

Summary

• C has three pools of data memory (+ code memory)
  - Static storage: global variable storage, “permanent, entire program run
  - The Stack: local variable storage, parameters, return address
  - The Heap (dynamic storage): malloc() gets space from here, free() returns it

• Common (Dynamic) Memory Problems
  - Using uninitialized values
  - Accessing memory beyond your allocated region
  - Improper use of free by changing pointer handle returned by malloc
  - Memory leaks: mismatched malloc/ free pairs