CS 61C: Great Ideas in Computer Architecture
C Memory Management

Instructors:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

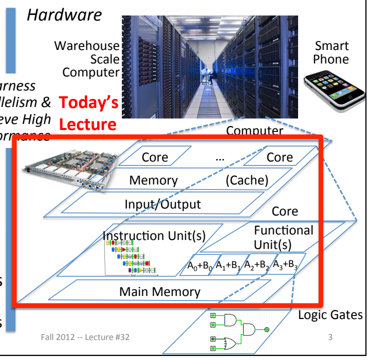
Review

- 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+M} blocks: 2^N ways x 2^M sets
- Multi-level caches
  - Optimize first level to be fast!
  - Optimize 2^{nd} and 3^{rd} levels to minimize the memory access penalty

You Are Here!

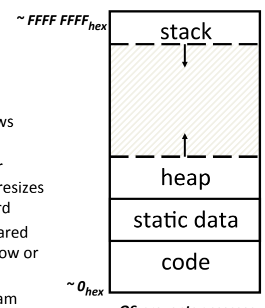
Software

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages



Recap: C Memory Management

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



OS prevents accesses between stack and heap (via virtual memory)

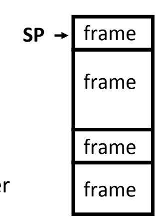
Recap: Where are Variables Allocated?

- If declared outside a procedure, allocated in "static" storage
- If declared inside procedure, allocated on the "stack" and freed when procedure returns
  - main() is treated like a procedure

```
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



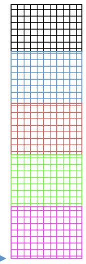
Recap: The Stack

- Last In, First Out (LIFO) data structure

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

Stack grows down

stack



Stack Pointer →

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 is tricky*: memory can be allocated / deallocated at any time

Managing the Heap

- C supports five functions for heap management: **`malloc()`**, `calloc()`, **`free()`**, `cfree()`, `realloc()`
- `malloc(n)`:
  - Allocate a block of uninitialized memory
  - NOTE: Subsequent calls need not yield blocks in continuous sequence
  - `n` is an integer, indicating size of allocated memory block in bytes
  - `sizeof` determines size of given type in bytes, produces more portable code
  - Returns a pointer to that memory location; NULL return indicates no more memory
  - Think of ptr as a *handle* that also describes the allocated block of memory; Additional control information stored in the heap around the allocated block!
- Example:

```
int *ip;
ip = malloc(sizeof(int));

struct treeNode *tp;
tp = malloc(sizeof(struct treeNode));
```


Managing the Heap

- `free(p)`:
  - Releases memory allocated by `malloc()`
  - `p` is pointer containing the address *originally* returned by `malloc()`

```
int *ip;
ip = malloc(sizeof(int));
... .. ..
free(ip);  /* Can you free(ip) after ip++ ? */

struct treeNode *tp;
tp = malloc(sizeof(struct treeNode));
  ... .. ..
free(tp);
```

  - When insufficient free memory, `malloc()` returns `NULL` pointer; **Check for it!**

```
if ((ip = malloc(sizeof(int))) == NULL){
  printf("\nMemory is FULL\n");
  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)!

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)

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

http://valgrind.org

Using Memory You Don't Own

- What is wrong with this code?

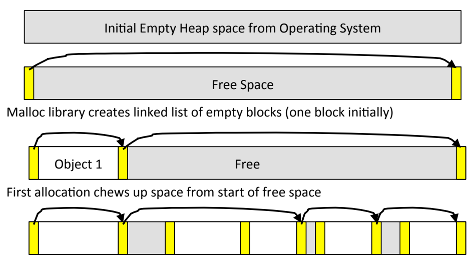
```
int *ipr, *ipw;
void ReadMem() {
      int i, j;
      *ipr = malloc(4 * sizeof(int));
       i = *(ipr - 1000); j = *(ipr + 1000);
      free(ipr);
   }

   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

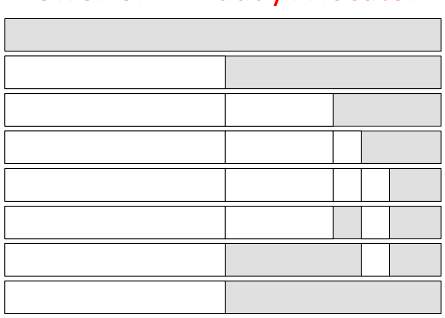


After many mallocs and frees, have potentially long linked list of odd-sized blocks
Frees link block back onto linked list – might merge with neighboring free space

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"



Malloc 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!

Administrivia

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++;
}
```


Faulty Heap Management

- What is wrong with this code?

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

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


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[128];
  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;
}
```

Using Memory You Don't Own

- What is wrong with this code?

```
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

- `calloc(n, size)`:
  - Allocate n elements of same data type; n can be an integer variable, use `calloc()` to allocate a dynamically size array
  - `n` is the # of array elements to be allocated
  - `size` is the number of bytes of each element
  - `calloc()` guarantees that the memory contents are initialized to zero

  E.g.: allocate an array of 10 elements

  ```
  int *ip;
  ip = calloc(10, sizeof(int));
  ```

  `*(ip+1)` refers to the 2nd element, like `ip[1]`

  `*(ip+i)` refers to the i+1th element, like `ip[i]`

  Beware of referencing beyond the allocated block: e.g., `*(ip+10)`
  - `calloc()` returns `NULL` if no further memory is available
- `cfree(p) // Legacy function – same as free`
  - `cfree()` releases the memory allocated by `calloc()`; E.g.: `cfree(ip);`

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!

  E.g.: allocate an array of 10 elements, expand to 20 elements later

  ```
  int *ip;
  ip = malloc(10*sizeof(int));
  /* always check for ip == NULL */
  … … …
  ip = realloc(ip,20*sizeof(int));
  /* always check for ip == NULL */
  /* contents of first 10 elements retained */
  … … …
  realloc(ip,0); /* identical to free(ip) */
  ```


Using Memory You Don't Own

- What is wrong with this code?

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

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;
}
```

Student Roulette

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