CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Dynamic Memory Management

Instructors:
Randy H. Katz
David A. Patterson

http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda

• C Memory Management
• Administrivia
• Technology Break
• Common Memory Problems

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

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

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

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

Recap: The Stack

- Last in, First Out (LIFO) data structure

Managing the Heap

- C supports functions for heap management: `malloc()`, `calloc()`, `free()`, `calloc()`, `realloc()`
  - `malloc(size)`
  - Allocates a block of uninitialized memory
  - `calloc(size)`
  - Returns the size of given type in bytes, produces more portable code
  - Additional control information stored in the heap around the allocated block

Example:

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

Peer Instruction: What’s Wrong with this Code?

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

Red: printf overwrites stack frame

```
main () {
    int *stackAddr, content;
    stackAddr = ptr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    printf("%d", content); /* 13451514 */
}
```

Peer Instruction: What’s Wrong with this Code?

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

```
main () {
    int *stackAddr, content;
    stackAddr = ptr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    printf("%d", content); /* 13451514 */
}
```

Red: printf overwrites stack frame

Never return pointers to local variable from functions

Your compiler will warn you about this – don’t ignore such warnings!

Managing the Heap

- `free()`:
  - `free()` releases memory allocated by `malloc()`
  - `p` is pointer containing the address originally returned by `malloc()`
    - `i` is pointer containing the address originally returned by `malloc()`
      `i` is FULL
  - When insufficient free memory, `malloc()` returns NULL. Check for it!
    - If `ip = malloc(sizeof(int));` is 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!
Managing the Heap

- \texttt{calloc(n, size)}:
  - Allocate \( n \) elements of same data type; \( n \) can be an integer variable, use \texttt{calloc()} to allocate a dynamically size array
  - \( n \) is the \# of array elements to be allocated
  - \texttt{size} is the number of bytes of each element
  - \texttt{calloc()} guarantees that the memory contents are initialized to zero
  - E.g.: allocate an array of 10 elements
    \begin{verbatim}
    int *ip;
    ip = calloc(10, sizeof(int));
    *(ip+1) refers to the 2nd element, like ip[1]
    *(ip+i) refers to the \( i \)th element, like ip[i]
    \end{verbatim}
  - \texttt{calloc()} returns NULL if no further memory is available

- \texttt{realloc(p, size)}:
  - Resize a previously allocated block at \( p \) to a new size
  - If \( p \) is NULL, then \texttt{realloc} behaves like \texttt{malloc}
  - If \( size \) is 0, then \texttt{realloc} behaves like \texttt{free}, deallocates 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
    \begin{verbatim}
    int *ip;
    /* always check for ip == NULL */
    ip = realloc(ip,20*sizeof(int)); /* contents of first 10 elements retained */
    realloc(ip,0); /* identical to free(ip) */
    \end{verbatim}

Agenda

- C Memory Management
- Administrivia
- Technology Break
- Example

Administrivia

- Project 4: Single Cycle Processor in Logisim
  - Due Sat 27 Nov (fine to submit early!)
  - Face-to-Face grading in Lab, Th 2 Dec
- EC: Performance Improvements to Proj #3 (due Tu 29 Nov)
- Final Review Sessions:
  - Mon 6 Dec, 1400-1700, 10 Evans
- Final: Mon 13 Dec 0800-1100 (220 Hearst Gym)
  - Like midterm: T/F, M/C, short answers
  - Whole Course: readings, lectures, projects, labs, hw
  - Emphasize 2nd half of 61C + midterm mistakes
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)

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 analyses 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
• You will be introduced in Valgrind in Lab #6!

Using Uninitialized Values

• What is wrong with this code?

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

• What is wrong with this code?

```c
void foo(int *pi) {
    int j;
    *pi = j;
    /* j is uninitialized, copied into *pi */
}
void bar() {
    int i=10;
    foo(&i);
    printf("i = %d\n", i);
    /* Using i, which now has junk value */
}
```

Valgrind Output (Highly Abridged!)

```
==98863== Memcheck, a memory error detector
==98863== Using Valgrind-3.6.0 and LdVEX, rerun with -h for copyright info
==98863== Conditional [emp or move depends on uninitialised value(s)]
==98863== at 0x10001A1E: __xprind (in /usr/lib/libSystem.B.dylib)
==98863== by 0x10007B8A: __xprind_l (in /usr/lib/libSystem.B.dylib)
==98863== by 0x1000A1A4: printf (in /usr/lib/libSystem.B.dylib)
==98863== by 0x1000000E: main(slide21.c:13)
==98863== Uninitialised value was created by a stack allocation
==98863== at 0x10000EB0: foo (slide21.c:3)
==98863==
```

Valgrind Output (Highly Abridged!)

```
==98863== HEAP SUMMARY:
==98863== in use at exit: 4,184 bytes in 2 blocks
==98863== total heap usage: 2 allocs, 0 frees, 4,184 bytes allocated
==98863== LEAK SUMMARY:
==98863== definitely lost: 0 bytes in 0 blocks
==98863== indirectly lost: 0 bytes in 0 blocks
==98863== possibly lost: 0 bytes in 0 blocks
==98863== still reachable: 4,184 bytes in 2 blocks
==98863== suppressed: 0 bytes in 0 blocks
==98863== Reachable blocks (those to which a pointer was found) are not shown.
==98863== To see them, rerun with: --leak-check=full --show-reachable=yes.
```
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;
}
```

Using Memory You Don’t Own

• Following a NULL pointer to mem addr 0!

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

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
        /* What if head happens to be NULL? */
        head = head->next;
    }
    return head->val; /* What if head is NULL? */
}
```

Using Memory You Don’t Own

• What is wrong with this code?

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

void WriteMem() {
    *ipw = malloc(5 * sizeof(int));
    *(ipw - 1000) = 0; *(ipw + 1000) = 0;
    free(ipw);
}
```

Using Memory You Don’t Own

• Using pointers beyond the range that had been malloc’d

— May look obvious, but what if mem ref had been result of pointer arithmetic that erroneously took them out of the allocated range?

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

void WriteMem() {
    *ipw = malloc(5 * sizeof(int));
    *(ipw - 1000) = 0; *(ipw + 1000) = 0;
    free(ipw);
}
```

Using Memory You Don’t Own

• What is wrong with this code?

```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;
}

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;
}
```

Using Memory You Don’t Own

• Improper matched usage of mem handles

```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;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    /* oops, forgot! */ fib = 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;
}
```

Remember: realloc may move entire block

What if array is moved to new location?
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 (; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
    result[i] = s1[j];
  }
  for (; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
    result[i] = s2[j];
  }
  result[++i] = ‘\0’;
  return result;
}

Using Memory You Don’t Own

• Beyond stack/read/write

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

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
”, str);
}

Using Memory You Haven’t Allocated

• Reference beyond array bounds

void StringManipulate() {
  const char *name = “Safety Critical”;
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = ‘\0’;
  /* Write Beyond Array Bounds */
  printf(“%s”, str);
  /* Read Beyond Array Bounds */
}

Faulty Heap Management

• What is wrong with this code?

int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  /* Allocate memory for pi */
  free(pi);
}

Faulty Heap Management

• Memory leak: more mallocs than frees

int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  /* Allocate memory for pi */
  /* Oops, leaked the old memory pointed to by pi */
  free(pi);
  /* foo() is done with pi, so free it */
}

void main() {
  pi = malloc(4*sizeof(int));
  foo(); /* Memory leak: foo leaks it */
}
Faulty Heap Management

• What is wrong with this code?

int *plk = NULL;
void genPLK() {
    plk = malloc(2 * sizeof(int));
    plk++;
}

Faulty Heap Management

• Potential memory leak – handle has been changed, do you still have copy of it that can correctly be used in a later free?

int *plk = NULL;
void genPLK() {
    plk = malloc(2 * sizeof(int));
    plk++; /* Potential leak: pointer variable incremented past beginning of block */
}

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);
}

Faulty Heap Management

• Can’t free non-heap memory; Can’t free memory that hasn’t been allocated

void FreeMemX() {
    int fnh = 0;
    free(&fnh); /* Oops! freeing stack memory */
}
void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1); /* fum+1 is not a proper handle; points to middle of a block */
    free(fum);
    free(fum);
    /* Oops! Attempt to free already freed memory */
}

Summary

• C has three pools of data memory (+ code memory)
  – Static storage: global variable storage, basically permanent, entire program run
  – The Stack: local variable storage, parameters, return address
  – The Heap (dynamic storage): malloc()/calloc() grabs space from here, free()/cfree() returns it, realloc() resizes space

• Common (Dynamic) Memory Problems
  – Using uninitialized values
  – Accessing memory beyond your allocated region
  – Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
  – Memory leaks: mismatched malloc/free pairs