CS61C : Machine Structures

Lecture 2.1.1
Memory Management

2004-06-28

Kurt Meinz

inst.eecs.berkeley.edu/~cs61c

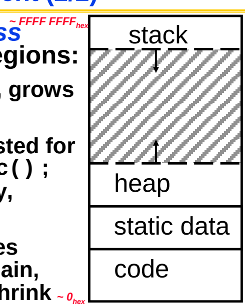
Memory Management (1/2)

- **Variable declaration allocates memory**
  - **outside a procedure -> static storage**
  - **inside procedure -> stack**
    - freed when procedure returns.
- **Malloc request**
  - **Pointer: static or stack**
  - **Content: on heap**

```
int myGlobal;
main() {
 int myTemp;
 int *f=
   malloc(16);
}
```


Memory Management (2/2)

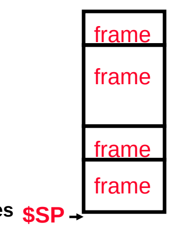
- **A program's *address space* contains 4 regions:**
  - **stack: proc frames, grows downward**
  - **heap: space requested for pointers via `malloc()` ; resizes dynamically, grows upward**
  - **static data: variables declared outside main, does not grow or shrink**
  - **code: loaded when program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



For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

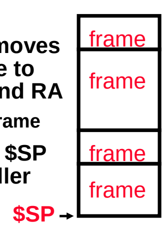
The Stack (1/4)

- **Environment frames from 61a ➔**
  - **Really exist in memory**
  - **Laid out using stack structure from 61b**
- **Terminology:**
  - **Stack is composed of frames**
  - **A frame corresponds to one procedure invocation**
  - **Stack frame includes:**
    - Return address of caller
    - Space for other local variables



The Stack (2/4)

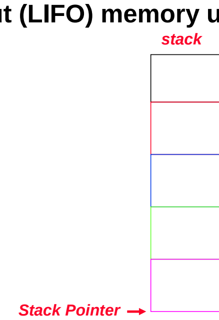
- **Implementation:**
  - **By convention, stack grows down in memory.**
  - **Stack pointer ($SP) points to next available address**
  - **PUSH: On invocation, callee moves $SP down to create new frame to hold callee's local variables and RA**
    - (old SP – new SP) ➔ size of frame
  - **POP: On return, callee moves $SP back to original, returns to caller**



The Stack (3/4)

- **Last In, First Out (LIFO) memory usage**

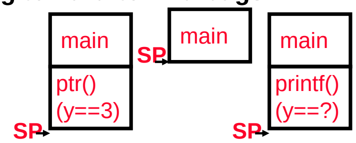
```
main ()
{ a(0);
}
  void a (int m)
  { b(1);
  }
    void b (int n)
    { c(2);
    }
      void c (int o)
      { d(3);
      }void d (int p)
       {
       }
```



The Stack (4/4): Dangling Pointers

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs !

```
int *ptr () {
   int y;
   y = 3;
   return &y;
 }
main () {
   int *stackAddr;
   stackAddr = ptr();
   printf("%d", *stackAddr); /* 3 */
   printf("%d", *stackAddr); /* XXX */
}
```



Static and Code Segments

- Code (Text Segment)
  - Holds instructions to be executed
  - Constant size
- Static Segment
  - Holds global variables whose addresses are known at compile time
    - Cf. Heap (malloc calls) where address isn't known

The Heap (Dynamic memory)

- Large pool of memory, not allocated in contiguous order
  - back-to-back requests for heap memory could result blocks very far apart
  - where Java `new` command allocates memory
- In C, specify number of bytes of memory explicitly to allocate item

```
int *ptr;
ptr = (int *) malloc(4);
/* malloc returns type (void *),
so need to cast to right type */
```

  - `malloc()`: Allocates raw, uninitialized memory from heap

Memory Management

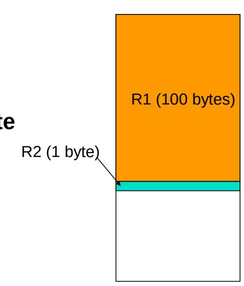
- How do we manage memory?
- Code, Static storage are easy: they never grow or shrink
- Stack space is also easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky: memory can be allocated / deallocated at any time

Heap Management Requirements

- Want `malloc()` and `free()` to run quickly.
- Want minimal memory overhead
- Want to avoid *fragmentation* – when most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

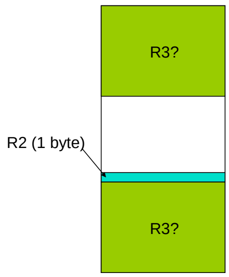
Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



K&R Malloc/Free Implementation

- From Section 8.7 of K&R
  - Code in the book uses some C language features we haven't discussed and is written in a very terse style, don't worry if you can't decipher the code
- Each block of memory is preceded by a header that has two fields:
  size of the block and
  a pointer to the next block
- All free blocks are kept in a linked list, the pointer field is unused in an allocated block

K&R Implementation

- `malloc()` searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system.
- `free()` checks if the blocks adjacent to the freed block are also free
  - If so, adjacent free blocks are merged (coalesced) into a single, larger free block
  - Otherwise, the freed block is just added to the free list

Choosing a block in `malloc()`

- If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?
  - best-fit: choose the smallest block that is big enough for the request
  - first-fit: choose the first block we see that is big enough
  - next-fit: like first-fit but remember where we finished searching and resume searching from there

Tradeoffs of allocation policies

- Best-fit: Tries to limit fragmentation but at the cost of time (must examine all free blocks for each malloc). Leaves lots of small blocks (why?)
- First-fit: Quicker than best-fit (why?) but potentially more fragmentation. Tends to concentrate small blocks at the beginning of the free list (why?)
- Next-fit: Does not concentrate small blocks at front like first-fit, should be faster as a result.

Administrivia (1/2)

- HW Grading:
  - Submit by 8pm Sundays
  - Solutions on Monday
  - Graded by Tuesday Lecture
  - Sign up for f2f in lab
    - <40/100: 30 Minutes
    - <90/100: 15 Minutes
    - Bring paper copy and understanding of what you got wrong.
    - Reader will give you points for your demonstration of better understanding/effort. (up to 90/100)

Administrivia (2/2)

- Projects will be similar
- No cheating the system!
  - You have to earn the points back.
  - Getting points back is dependent on you trying on the initial submission.
    - If you submit garbage thinking that you'll get all the points back in f2f, you are wrong!
- Office Hours 1- 2pm I House

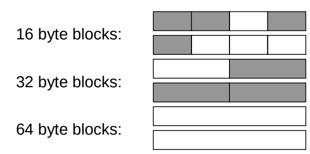
Slab Allocator

- A different approach to memory management (used in GNU `libc`)
- Divide blocks in to "large" and "small" by picking an arbitrary threshold size. Blocks larger than this threshold are managed with a freelist (as before).
- For small blocks, allocate blocks in sizes that are powers of 2
  - e.g., if program wants to allocate 20 bytes, actually give it 32 bytes

Slab Allocator

- Bookkeeping for small blocks is relatively easy: just use a *bitmap* for each range of blocks of the same size
- Allocating is easy and fast: compute the size of the block to allocate and find a free bit in the corresponding bitmap.
- Freeing is also easy and fast: figure out which slab the address belongs to and clear the corresponding bit.

Slab Allocator



16 byte block bitmap: 11011000

32 byte block bitmap: 0111

64 byte block bitmap: 00

Slab Allocator Tradeoffs

- Extremely fast for small blocks.
- Slower for large blocks
  - But presumably the program will take more time to do something with a large block so the overhead is not as critical.
- Minimal space overhead
- No fragmentation (as we defined it before) for small blocks, but still have wasted space!

Internal vs. External Fragmentation

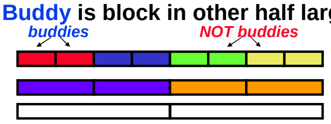
- With the slab allocator, difference between requested size and next power of 2 is wasted
  - e.g., if program wants to allocate 20 bytes and we give it a 32 byte block, 12 bytes are unused.
- We also refer to this as fragmentation, but call it *internal* fragmentation since the wasted space is actually within an allocated block.
- External fragmentation: wasted space between allocated blocks.

Buddy System

- Yet another memory management technique (used in Linux kernel)
- Like GNU's "slab allocator", but only allocate blocks in sizes that are powers of 2 (internal fragmentation is possible)
- Keep separate free lists for each size
  - e.g., separate free lists for 16 byte, 32 byte, 64 byte blocks, etc.

Buddy System

- If no free block of size n is available, find a block of size 2n and split it in to two blocks of size n
- When a block of size n is freed, if its neighbor of size n is also free, combine the blocks in to a single block of size 2n
  - Buddy is block in other half larger block



- Same speed advantages as slab allocator

Allocation Schemes

- So which memory management scheme (K&R, slab, buddy) is best?
  - There is no single best approach for every application.
  - Different applications have different allocation / deallocation patterns.
  - A scheme that works well for one application may work poorly for another application.

Automatic Memory Management

- Dynamically allocated memory is difficult to track – why not track it automatically?
- If we can keep track of what memory is in use, we can reclaim everything else.
  - Unreachable memory is called *garbage*, the process of reclaiming it is called *garbage collection*.
- So how do we track what is in use?

Tracking Memory Usage

- Techniques depend heavily on the programming language and rely on help from the compiler.
- Start with all pointers in global variables and local variables (root set).
- Recursively examine dynamically allocated objects we see a pointer to.
  - We can do this in constant space by reversing the pointers on the way down
- How do we recursively find pointers in dynamically allocated memory?

Tracking Memory Usage

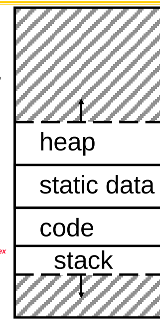
- Again, it depends heavily on the programming language and compiler.
- Could have only a single type of dynamically allocated object in memory
  - E.g., simple Lisp/Scheme system with only cons cells (61A's Scheme not "simple")
- Could use a *strongly typed* language (e.g., Java)
  - Don't allow conversion (casting) between arbitrary types.
  - C/C++ are not strongly typed.
- Here are 3 schemes to collect garbage

Bonus Slides

- The following material wasn't covered in lecture, but I leave it here for your enjoyment.

Intel 80x86 C Memory Management

- A C program's 80x86 *address space* :
  - **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
  - **stack**: local variables, grows downward



Linked List Example

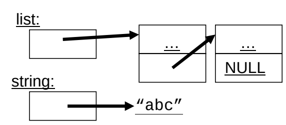
- Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

```
struct Node {
    char *value;
    struct Node *next;
};
typedef Node *List;

/* Create a new (empty) list */
List ListNew(void)
{ return NULL; }
```

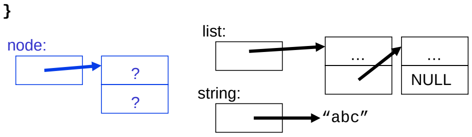

Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



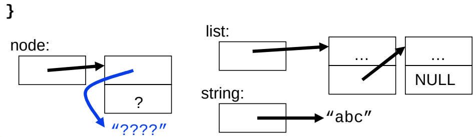
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



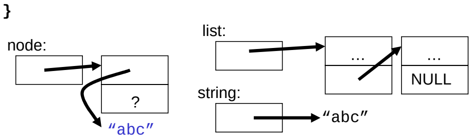
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



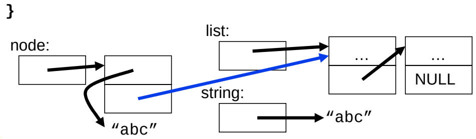
Linked List Example

```c
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Linked List Example

```c
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Linked List Example

```c
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```