CS61C – Machine Structures

Lecture 5 –Memory Mangement

9/7/2007

John Wawrzynek

(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/

Memory Allocation

- **Remember:**
  - Structure declaration does not allocate memory
  - Variable declaration does allocate memory
- **So far we have talked about several different ways to allocate memory for data:**
  1. Declaration at the beginning of a block
     `int i; struct Node list; char *string;`
  2. "Dynamic" allocation at runtime by calling allocation function (alloc).
     ```
     ptr = (struct Node *) malloc(sizeof(struct
        Node));
     ```
- **One more possibility exists:**

Memory Allocation

```
int myGlobal;
main() {

}
```

- Data declared outside of any procedure (before main).
- Similar to declaration at beginning of a block, but has "global" scope.

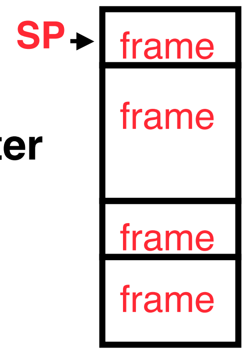
Where are these allocated?

- If declare outside a procedure, allocated in "static" storage
- If declare inside procedure, allocated on the "stack" and freed when procedure returns.
  - Note: `main()` is a procedure

```
int myGlobal;
main() {
  int myTemp;
}
```

The Stack

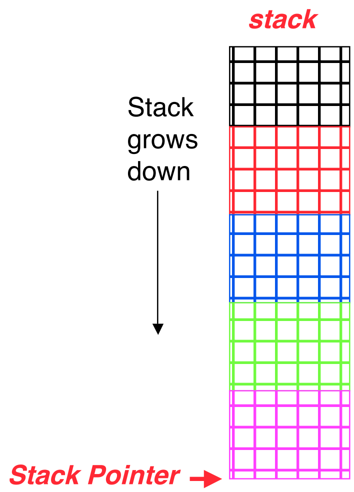
- Stack frame includes:
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- Stack frames form contiguous blocks of memory; stack pointer tells where top stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames



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)
        {
        }
```



C Memory Management

- C has 3 pools of memory
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address (location of "activation records" in Java or "stack frame" in C)
  - The Heap (dynamic storage): data lives until deallocated by programmer
- C requires knowing where objects are in memory, otherwise things don't work as expected
  - Java hides location of objects

The Heap (Dynamic memory)

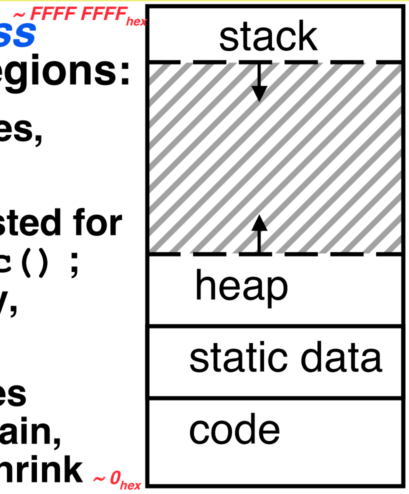
- Large pool of memory, not allocated in contiguous order
  - back-to-back requests for heap memory could result in blocks very far apart
  - where Java `new` command allocates memory
- In C, specify number of bytes of memory explicitly to allocate item

```
struct int *iptr;
iptr = (int *) malloc(8*sizeof(int));
/* malloc returns type (void *),
so need to cast to right type */
```

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

Typical C Memory Management

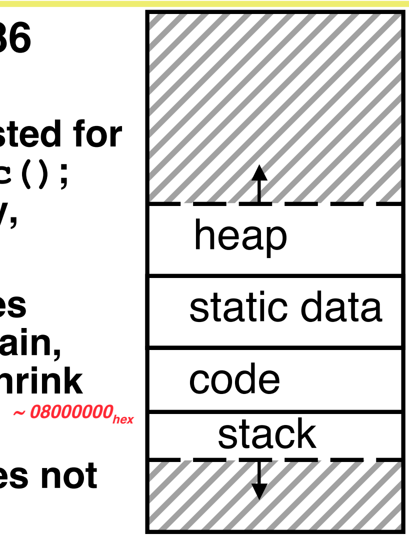
- A program's *address space* contains 4 regions:
  - **stack**: local variables, grows downward
  - **heap**: space requested for pointers via `malloc()` ; resizes dynamically, grows upward
  - **static data**: variables declared outside main, does not grow or shrink
  - **code**: loaded when program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



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

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



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

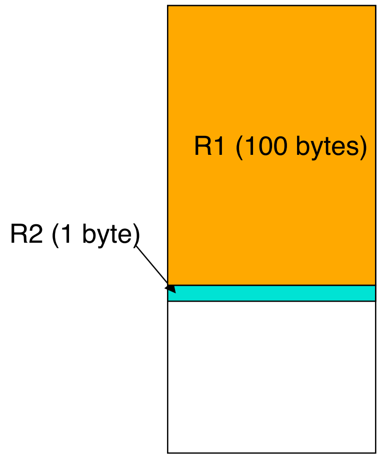
° Want to avoid *fragmentation** – when most of our free memory is in many small chunks

- 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.

* This is technically called *external fragmention*

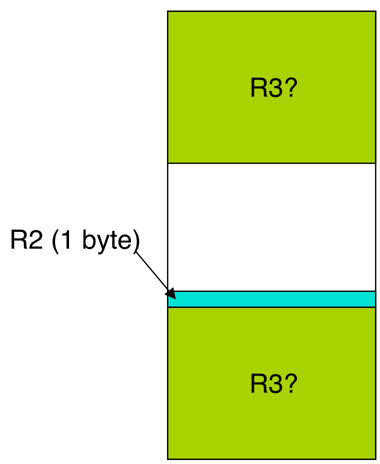
Heap Management

- 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

- Look at 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
- 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. If what it gets can't satisfy the request, it fails.
- `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

Peer Instruction – Pros and Cons of fits

A. The con of first-fit is that it results in many small blocks at the beginning of the free list

B. The con of next-fit is it is slower than first-fit, since it takes longer in steady state to find a match

C. The con of best-fit is that it leaves lots of tiny blocks

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

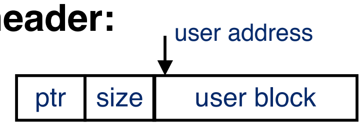
Tradeoffs of allocation policies

- **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.

Example Dynamic Memory Allocator

K&R malloc/free

- Each block begins with a header:



- Free list:
  - *Circularly linked*
  - Partially traversed by malloc and free
  - Block appear on list in increasing memory position
- *Next fit* algorithm for allocation
- New block taken from tail of next sufficiently large block
- Free merges blocks existing free block(s)

K&R Allocator

```
typedef struct header {
  struct header * ptr; /* next free block */
  unsigned size;     /* size of this block */
} Header;
```

- K&R uses a "union" type to force alignment. Complicates field extraction: `p->size` becomes `p->s.size`
  - For simplicity here, we don't use the union.
- Globals:

```
static Header base;
          /* empty list to get started */
static Header *freep = NULL;
          /* start of free list */
```

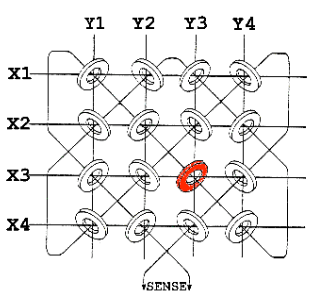

K&R malloc

```
void *malloc(unsigned n bytes)
{
  Header *p, *prevp;
  Header *morecore(unsigned);  /* used to get another large block from OS */
  unsigned nunits;

  nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) +1;
  /* round up to allocate in units of sizeof(Header) */

  if ((prevp = freep) == NULL) {   /* no free list yet */
    base.ptr = freep = prevp = &base;  base.size = 0;
  }
  for (p = prev->ptr; ; prevp = p,  p = p->ptr) {
    if (p->size >= nunits) {    /* big enough */
      if (p->size == nunits)       /* exactly */
        prevp->ptr = p->ptr;
      else {          /* allocate tail end */
        p->size -= nunits;  p+= p->size;  p->size = nunits;
      }
      freep = prevp;   /* start next search here next time */
      return (void *)(p + 1);  /* point past the header */
    }
    if (p == freep)    /* wrapped around free list */
      if ((p = morecore(nunits)) == NULL) return NULL;  /* none left */
  }
}
```

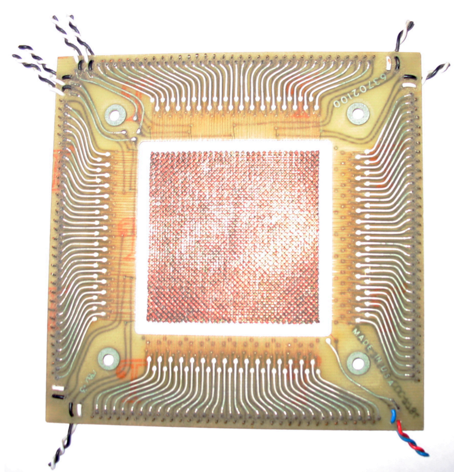
K&R free

```
void free(void *ap)  /* keeps blocks ordered by address */
{
  Header *bp, *p;

  bp = (Header *)ap - 1;    /* point to block header */

  for (p = freep;  !(bp > p && bp < p->ptr);  p = p->ptr)
    if (p >= p->ptr && (bp > p || bp < p->ptr)) break;
    /* freed block at start or end of arena */

  if (bp + bp->size == p->ptr) {    /* join to next block */
    bp->size += p->ptr->size;
    bp->ptr = p->ptr->ptr;
  }  else bp->ptr = p->ptr;
  if (p + p->size == bp) {      /* join to previous block */
    p->size += bp->size;
    p->ptr = bp->ptr;
  } else
    p->ptr = bp;
  freep = p;
}
```


"Core" memory? Some history.



Array of ferrite rings, "cores", threaded with wires for writing and reading.



64 x 64 array of cores

How much denser is modern semiconductor memory?

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?

° Techniques depend heavily on the programming language and rely on help from the compiler

And in conclusion…

° C has 3 pools of memory

- Static storage: global variable storage, basically permanent, entire program run
- The Stack: local variable storage, parameters, return address
- The Heap (dynamic storage): `malloc()` grabs space from here, `free()` returns it.

° `malloc()` handles free space with freelist. Three different ways to find free space when given a request:

- First fit (find first one that's free)
- Next fit (same as first, but remembers where left off)
- Best fit (finds most "snug" free space)

Extras

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

- **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 external fragmentation (as we defined it before) for small blocks, but still have wasted space!**

Internal vs. External Fragmentation

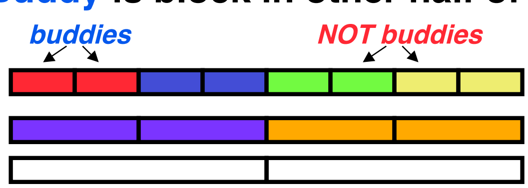
- **With the slab allocator, difference between requested size and next power of 2 is wasted**
  - **e.g., if program wants to allocate 20 bytes and we give it a 32 byte block, 12 bytes are unused.**
- **We also refer to this as fragmentation, but call it *internal* fragmentation since the wasted space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