Stack and Heap

Call Stack for Procedure Calls

- Each procedure (method/function) call pushes a new *frame* (a.k.a. *stack frame*) on a stack
- The frame contains space for all the locals, including arguments and return
- Also a pointer to previous "top of stack"
- For execution with variable-size stack frames, we need a *stack pointer* (top of stack) and a *frame pointer* (base of current stack frame)
- These are typically supported by the architecture (i.e., they are registers)

Call Stack for Procedure Calls

```
int foo(int i) {
  int j;
  j = bar(3,i);
  return j+baz(j);
}

int bar(int k, int l) {
  int m = k * k + 7 * l + 3;
  return m;
}

int baz(int n) { ... }
```

- What does the stack look like during the execution of *foo*?

The Stack as Memory

- The stack is a great way to remember things!
  - Automatically managed: no need to "free" data on the stack
  - Very efficient and fast, hardware-supported
- But: it only works when the lifetimes of data are hierarchical
  - Newer data should die before older ones
  - Data lifetimes are tied to procedure lifetimes
- For data that live longer, we have other structures: static data and the heap

Static Data: Different Kinds

```
int e;
void fun() {
  static char *root;  ...
}
class A {
  static int i;  ...
};
```

- Static data can be global or local: do not confuse namespace with lifetime
- The problem is that these are even more limited than the stack!
  - Static variables appear in the code
  - We know the number of static variables at compile-time!
  - Hence the name “static”

Heap

- The *heap* is the area to store data with (relatively) unknown lifetimes
- You already know that we manage this with *malloc/free* or *new/delete*
- What is the structure of the heap? How are *malloc/free* implemented?
- Main idea: get space from OS, manage it internally
  - Keep track of holes (from *free*)
  - Keep track of unallocated data
  - Keep other data structures for fast *malloc/free*

A Very Simple Allocator (K&R)

- Your C book had a dead-simple allocator (Chapter 8)
- Single free-list, ordered by address
- Header keeps size of allocated block
- Linear searches for both *malloc* and *free*

A Very Simple Allocator (Rewriting of K&R)

```c
#include <stdbool.h>
#include <unistd.h>

typedef long Align;        /* for alignment to
                              long boundary */
typedef union header {    /* block header */
  struct {
    union header *ptr;    /* next block if on
                              free list */
    size_t size;          /* size of this block */
  } s;
  Align x;               /* force alignment of blocks */
} Header;

static Header base = {0};     /* empty list to get
                                  started */
static Header* freep = NULL; /* start of free list */
```

A Very Simple Allocator (malloc–I)

```
void* kr_malloc (size_t nbytes) {
  Header*  p;
  Header*  prevp;
  size_t   nunits;

  nunits = 1 + (nbytes + sizeof(Header) - 1) /
                 sizeof(header);
  prevp = freep;
  if (prevp == NULL) {          /* no free list yet */
    base.s.ptr  = freep = prevp = &base;
    base.s.size = 0;
  }
```

A Very Simple Allocator (malloc–II)

```
  for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
    if (p->s.size >= nunits) {     /* big enough */
      if (p->s.size == nunits)     /* exactly */
        prevp->s.ptr = p->s.ptr;
      else {                     /* allocate tail end */
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits
      }
      freep = prevp;
      return p+1;
    }
    if (p == freep) {  /* wrapped around free list */
      p = morecore(nunits);
      if (p == NULL) return NULL; /* none left */
    }
  } /* for */
} /* kr_malloc */
```

A Very Simple Allocator (morecore)

```
#define NALLOC 1024

Header* morecore(unsigned int nu) {
  char*    cp;
  Header* up;

  if (nu < NALLOC) nu = NALLOC;
  cp = sbrk(nu * sizeof(Header));

  if (cp == (void *) -1)
    return NULL;
  up = (Header *) cp;
  up->s.size = nu;
  free((void *)(up+1);
  return freep;
}
```

A Very Simple Allocator (freep)

```
void free(void *ap) {
  Header* bp = (Header *)ap - 1;
  Header* p;
  for (p=freep; bp <= p || bp >= p->s.ptr; p=p->s.ptr)
    if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
      break; /* freed before beginning or after end */
  if (bp+bp->s.size == p->s.ptr) { /* coalesce next */
    bp->s.size += p->s.ptr->s.size;
    bp->s.ptr = p->s.ptr->s.ptr;
  } else
    bp->s.ptr = p->s.ptr;
  if (p+p->s.size == bp) { /* coalesce with prev */
    p->s.size += bp->s.size;
    p->s.ptr = bp->s.ptr;
  } else
    p->s.ptr = bp;
  freep = p;
}
```

What Do Realistic Allocators Do Differently?

- No linear search on *free*
  - no need to have list address-ordered
  - often coalescing done by keeping footers instead of just headers, so that the previous block's size is also known. More overhead per allocated block.
- Best-fit-like policies instead of first-fit
- Lots of size classes, more complex data structure than linked list
  - *bitmap allocation* for small objects
  - direct *mmap* for large ones
- Multi-threading support
  - for avoiding *malloc/free* bottlenecks
  - for avoiding *false sharing*
  - *huge* performance impact!