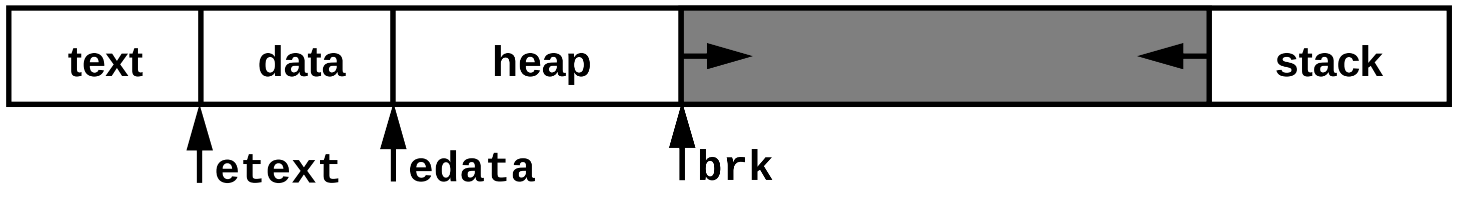
Dynamic Memory Allocation

- C library functions

```
void *malloc(size_t nbytes);          size_t is an unsigned type
void free(void *ptr);

struct foo *p = malloc(sizeof *p);
...
free(p);
```

- Memory layout



`brk(0100000)`	sets the end of the heap at address **`0100000`**
`sbrk(12)`	increments the end of the heap by 12 bytes

- Could use **`sbrk`** instead of **`malloc`**, e.g.,

```
p = (struct foo *)sbrk(sizeof *p); but
```

    it's inefficient
    it's not portable
    what about free?

A Fast, Simple Malloc without Free

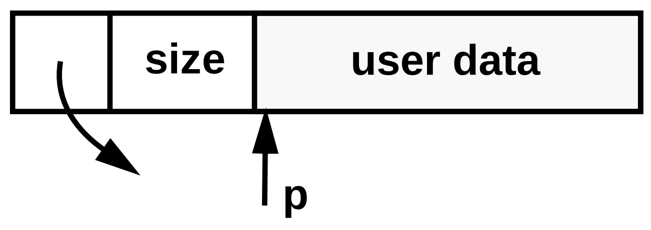
```
#define BLOCK 4096
extern void *sbrk(unsigned);
void *malloc(unsigned nbytes) {
   static int count = 0;
   static char *ptr = 0;
   union align { double d; unsigned u; void (*f)(void); } align;

     nbytes = (nbytes + (sizeof align - 1))&~(sizeof align - 1);
     if (nbytes <= count) {
         void *p = ptr;
         ptr += nbytes;
         count -= nbytes;
         return p;
     } else if (nbytes > BLOCK) {
         void *p = sbrk(nbytes);
         return p == (void *)-1 ? 0 : p;
     } else { /* count < nbytes <= BLOCK */
         void *p = sbrk(BLOCK);
         if (p == (void *)-1)
             return 0;
         count = BLOCK;
         ptr = p;
         return malloc(nbytes); /* no error check? */
     }
}
```

Allocation Algorithms

- First fit

  keep a linked list of free blocks

  search for the ***first*** one that's big enough

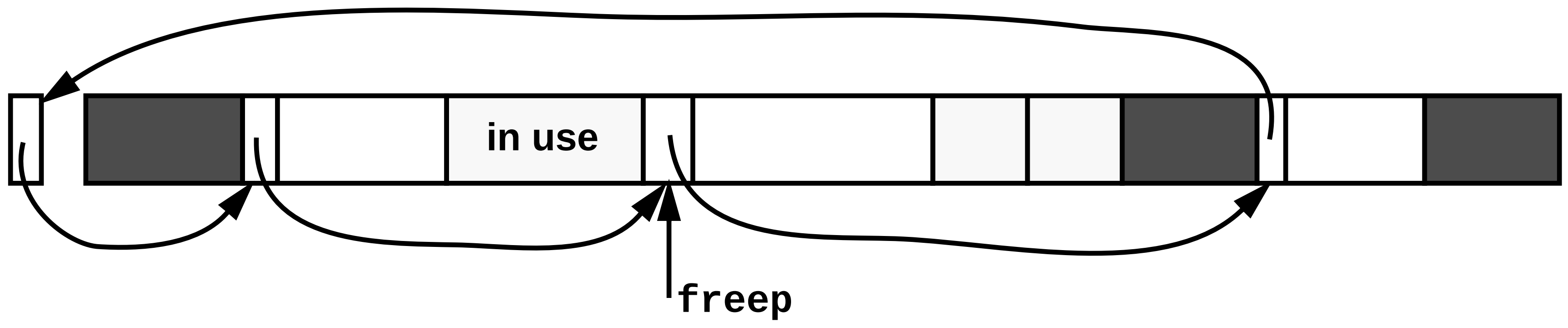


- Best fit

  keep a linked list of free blocks

  search for the ***smallest*** one that's big enough

- Free list: a circular list



- Free list is sorted in order of increasing addresses so that adjacent free blocks can be ***coalesced***

First Fit

```
static union header {
    struct {
        union header *link;
        unsigned size;
    } s;
    union align {
        double d; unsigned u; void (*f)(void);
    } x;
} freelist = { &freelist, 0 }, *freep = &freelist;
#define BLOCK 1024

void *malloc(unsigned nbytes) {
    union header *p, *q;
    unsigned size = (nbytes + sizeof (union header) - 1)/
                          sizeof (union header) + 1;
    <search for a block that is >= size units>
    if (size < BLOCK)
        size = BLOCK;
    p = sbrk(size*sizeof *p);
    if (p == (void *)-1)
        return 0;
    p->s.size = size;
    <insert this block after freep block in free list>
    return malloc(nbytes);
}
```

First Fit, cont'd

<search for a block that is>= `size` ***units>*** ≡

```
q = freep;
do {
    p = q->s.link;
    if (p->s.size > size) {
        p->s.size -= size;
        p += p->s.size;
        p->s.size = size;
        freep = q;
        return p + 1;
    } else if (p->s.size == size) {
        q->s.link = p->s.link;
        freep = q;
        return p + 1;
    }
    q = p;
} while (p != freep);
```

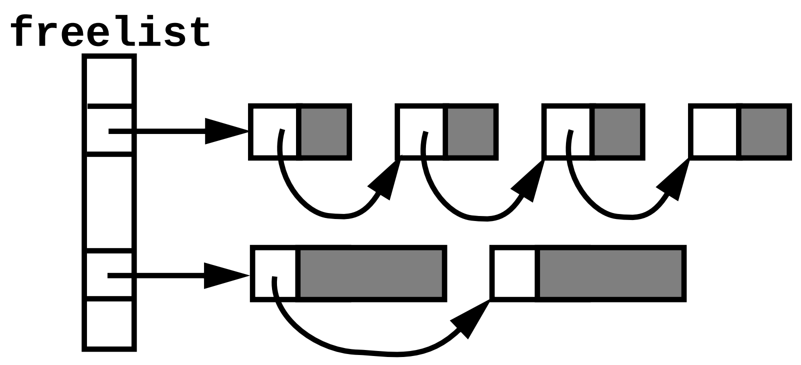
First Fit, cont’d

```
void free(void *ptr) {
    union header *bp = (union header *)ptr - 1, *p;
    if (ptr == 0)
        return;
    for (p = freep; ; p = p->s.link)
        if (bp > p && bp < p->s.link
        || p >= p->s.link && (bp > p || bp < p->s.link))
            break;
    if (bp + bp->s.size == p->s.link) {
        bp->s.size += p->s.link->s.size;
        bp->s.link = p->s.link->s.link;
    } else
        bp->s.link = p->s.link;
    if (p + p->s.size == bp) {
        p->s.size += bp->s.size;
        p->s.link = bp->s.link;
    } else
        p->s.link = bp;
    freep = p;
}
```

- What are the pros and cons of first fit?
- see K&R Section 8.7 and **`src/malloc/firstfit.c`**

Quick Fit

- “Quick fit” special-cases small allocations



```
static union header {...} *freelist[32];

void *malloc(unsigned nbytes) {
    union header *p;
    unsigned size = (nbytes + sizeof (union header) - 1)/
                       sizeof (union header) + 1;

    if (size < sizeof freelist/sizeof freelist[0]
    && (p = freelist[size]) != NULL) {
        freelist[size] = p->s.link;
        return p + 1;
    } else
        return firstfit_malloc(nbytes);
}
```

Quick Fit, cont'd

```
void free(void *ptr) {
    union header *bp = (union header *)ptr - 1;

    if (ptr == NULL)
        return;
    if (bp->s.size < sizeof freelist/sizeof freelist[0]) {
        bp->s.link = freelist[bp->s.size];
        freelist[bp->s.size] = bp;
    } else
        firstfit_free(ptr);
}
```

see **src/malloc/quickfit.c**

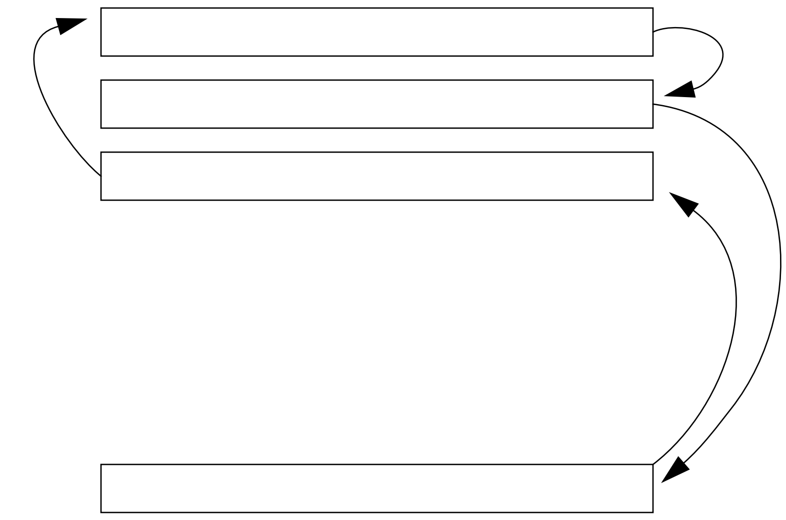
Fast Allocation Techniques

- What if your allocation sizes are not supported by “malloc” and “free” as a special case (say you are dealing with lines in an editor)?
- Build a private allocation module “pmalloc” and “pfree”

  `pmalloc() gets a line from free list`

  `pfree() puts a line into the free list`

  -



- **“pmalloc” and “pfree” take exactly one operation with the FreeList**
- **They can be built on top of “malloc” and “free.” How?**

Fast Allocation Techniques, cont'd

- What if we have to make the general case fast?
- free() merges freed memory to reduce fragmentation; needs to know:

  Are my adjacent pieces of memory free or not?
- **The worst case in the previous methods is to traverse the whole free list**
- Idea 1: Use a single-bit tag to indicate whether the memory is free

```
static union Header {
    struct {
        union Header *link;
        unsigned size: 31;
        unsigned freed: 1;
    } s;
    union Align {
        double d; unsigned u; void (*f)(void);
    } x;
}
```

  malloc() and free() will clear and set the "freed" bit

  Use the memory being freed to find out the header of the next memory

  Check the "free" bit of the next memory to see whether it is freed

  What are the pros and cons of this method?
- **What about the previous memory since we don't know its size?**

Fast Allocation Techniques, cont’d

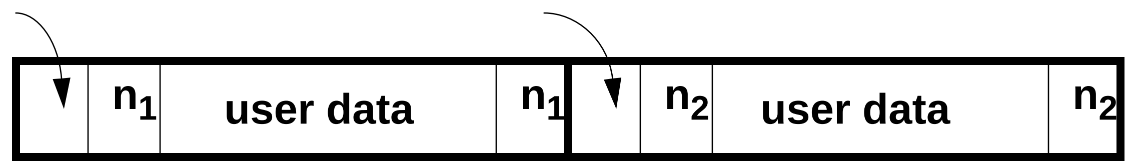
- **What about the state of the previous memory?**

  The main difficulty is not knowing the size of the previous memory

- **Idea 2: Use a “footer” for each memory and store the size**

  malloc() will set the “footer” in addition to the header

-



  Now we can tell the states of both adjacent memories fast

- **Do you also need a doubly-linked free list?**
- **For more information about memory allocation, see**

  D. Knuth, The Art of Programming