Agenda

Review: Pointers and Stack Diagrams

malloc/free

Warmup: Pointers

What is a pointer?
Which variables are pointers in this program?
What does the * and & syntax mean?
Draw the stack diagram.

```
int mystery(int* x, int* y) {
  int xx = *x;
  int yy = *y;
  xx++;
  yy++;
  *x = xx;
  *y = yy;
  return xx + yy;
}

int main() {
  int a = 2;
  int b = -4;
  int* ptrA = &a;
  int* ptrB = &b;

  int c = mystery(ptrA, ptrB);
  // stack diagram here
}
```

Warmup: Pointers

What is the relationship between arrays and pointers in C?
Draw the stack diagram.

```
int mystery(int* x, int* y) {
  for (int i = 0; i < 5; i++) {
    y[i] = *x;
  }
  return 0;
}

int main() {
  int* ptrA = malloc(sizeof(int) * 1);
  int* ptrB = malloc(sizeof(int) * 5);

  *ptrA = 10;
  int c = mystery(ptrA, ptrB);
  // stack diagram here

  free(ptrA);
  free(ptrA);
}
```

Review: Pointer Arithmetic

Recommended reference: Dive into Systems, Section 2.9.4

Pointer arithmetic provides the ability to move forward or backward in memory with respect to a pointer value

Example: `ptr + 1` moves forward in memory by the size of the data type of ptr

Useful in a few niche cases, specifically, working with generic blocks of memory

applications: databases, network messages, malloc

IMPORTANT

When adding/subtracting offsets to pointers, we increment/decrement based on the pointer's type

Example: Suppose the address of x is 0x4c568000

```
1) int* x = &a; x++;
2) char* x = &c; x++;
3)
struct m {
   int q;
   char buff[4];
};
struct m* x = &data; x++;
```

Draw the stack diagram for this program

```
int vals[4] = {1,2,3,4};
int* valptr = vals;

int v1 = vals[2];
int* v1ptr = valptr + 2;
int v2 = *v1ptr;
printf("%d %d\n", v1, v2);

for (int i = 0; i < 4; i++) {
  printf(“%d\n", vals[i]);
}

for (int* ptr = vals; ptr < vals+4; ptr++) {
  printf(“%p %d\n", ptr, *ptr);
}
```

Draw memory layout of vals

```
int vals[4] = {1,2,3,4};
int* valptr = vals;

int v1 = vals[2];
int* v1ptr = valptr + 2;
int v2 = *v1ptr;
printf("%d %d\n", v1, v2);
// draw stack diagram here

for (int i = 0; i < 4; i++) {
  printf("%d\n", vals[i]);
}

for (int* ptr = vals; ptr < vals+4; ptr++) {
  printf("%p %d\n", ptr, *ptr);
}
```

Suppose the beginning of the array is at location 0xffffeea0. What is the location of vals[0], vals[1], vals[2], etc. What is the location of valptr+1, valptr+2, etc?

Draw the stack diagram for this program

```
struct chunk {
  int size;
  struct chunk *next;
};

int main() {
  int size = sizeof(int) * 5;
  void *memory = malloc(size + sizeof(struct chunk));
  if (memory == NULL) {
    return 1;
  }

  struct chunk *cnk = (struct chunk*) memory;
  cnk->size = size;
  void* data = (void*) (cnk + 1);
  int* array = (int*) data;

  for (int i = 0; i < 5; i++) {
    array[i] = i;
  }
```

Malloc/Free

library calls, **not** system calls!

`malloc` and `free` can be replaced with our own implementation

Applications: Optimizing allocation strategies (speed, or size) so they are tailored to a specific application

sbrk

```
void *sbrk(intptr_t increment);
```

- **Program break**: the location of the end of the uninitialized data segment (a pointer to program memory)
- **sbrk** changes the location of the **program break** by the given number of bytes
- Increasing the program break allocates more memory to the process
- If successful, returns the previous program break.
  - e.g. the start of the newly allocated memory
- On error, the value `(void*) -1` is returned

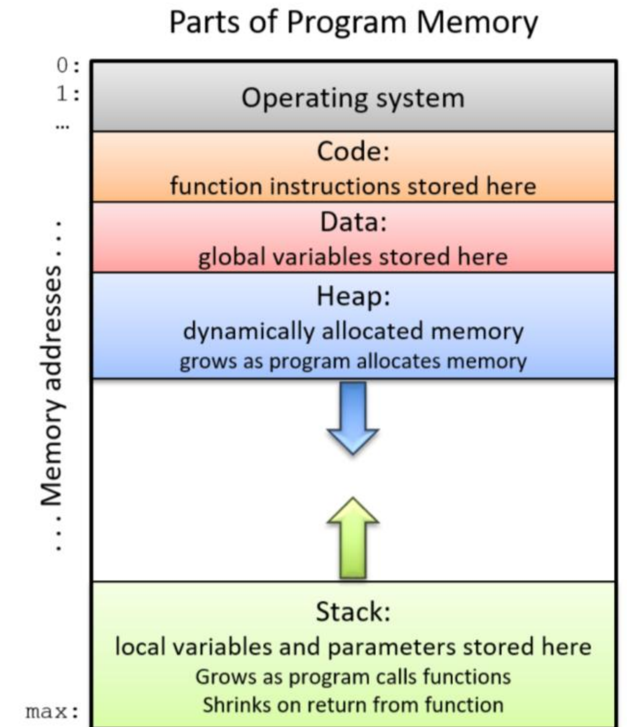

Figure 1. The parts of a program's address space.

mmap

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

DESCRIPTION
mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping (which must be greater than 0).

If addr is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping.

EXAMPLE

```
mem_block = mmap(NULL, sizeOfRegion,
        PROT_READ | PROT_WRITE,
        MAP_ANON | MAP_PRIVATE,
        -1, 0);
```

Malloc specification

```
$ man malloc
```

```
The malloc() function allocates size
bytes and returns a pointer to the
allocated memory. The memory is not
initialized. If size is 0, then malloc()
returns either NULL, or a unique pointer
value that can later be successfully
passed to free().
```

Free specification

```
$ man free
```

```
The free() function frees the memory
space pointed to by ptr, which must have
been returned by a previous call to
malloc(), calloc(), or realloc().
Otherwise, or if free(ptr) has already
been called before, undefined behavior
occurs. If ptr is NULL, no operation is
performed.
```

A simple implementation

```
void *mymalloc (size_t size) {
  if (size == 0){
    return NULL;
  }
  void *memory = sbrk(size);
  if (memory == (void *) -1) {
    return NULL;
  } else {
    return memory;
  }
}

void myfree(void *memory) {
  return;
}

void main() {
   char* c = mymalloc(sizeof(char)*10);
   myfree(c);
}
```

What is size_t?

Notice the void*, why do we need the cast?

What is this program doing?

What are the limitations of this implementation?

A better implementation

```
struct chunk {
  int size;
  struct chunk *next;
};
struct chunk *flist = NULL;
```

Maintain a list to freed memory so it can be re-used

Allocate enough space for both the header (struct chunk) and the requested memory

In malloc, check the free list for previously allocated memory.

- If none is found, use sbrk

In free, add memory to a list so it can be reused

A “first fit” implementation

```
void *mymalloc (size_t size) {
  if (size <= 0) return NULL;

  struct chunk *ptr = flist; //flist is global
  struct chunk *prev = NULL;
  while (ptr != NULL) {
    if (ptr->size >= size) {
      if (prev != NULL) {
        prev->next = ptr->next;
      } else {
        flist = ptr->next;
      }
      return (void*) (ptr + 1);
    } else {
      prev = ptr;
      ptr = ptr->next;
    }
  }
```

```
  void *memory = sbrk(size + sizeof(struct chunk));
  if (memory == (void *) -1) {
    return NULL;
  }
  else {
    struct chunk *cnk = (struct chunk*) memory;
    cnk->size = size;
    cnk->next = NULL;
    return (void*) (cnk + 1);
  }
}
```

New free

```
void myfree(void *memory) {
  if (memory != NULL) {
    struct chunk *cnk = (struct chunk*) ((struct chunk*) memory -1);
    cnk->next = flist;
    flist = cnk;
  }
  return;
}
```

Exercise: Visualize this program

```
struct test {
    float f;
    char msg[16];
};

void main() {
    struct test*  ptr = (struct test*) mymalloc(sizeof(test));
    myfree(ptr);
}
```

Your next homework: Memory Manager

Use mmap to request a large block of memory

malloc allocates chunks from this block (no sbrk)

Use a “worst fit” strategy

- Find largest chunk
- Break it into pieces: one piece goes to the user, the other remains in the free list
- Support coalesce option in free that recombines adjacent free memory into large chunks