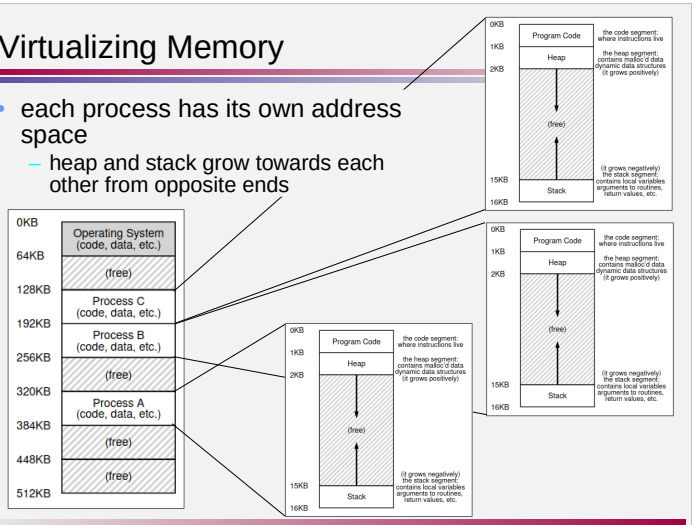
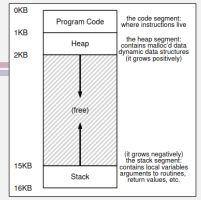
Virtualizing Memory

- each process has its own address space
  - heap and stack grow towards each other from opposite ends



Memory API



- two types of memory, distinguished by how allocations and deallocations are managed
  - *stack* memory is managed by the compiler
    - memory usage known at compile time

```
void func() {
  int x; // declares an integer on the stack
  ...
}
```

the compiler allocates memory on the stack when `x` is declared

the memory is deallocated when `x` goes out of scope (in this case, when the function returns)

  - *heap* memory is managed by the programmer
    - memory usage is only known at runtime

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

`malloc` results in memory being allocated on the heap

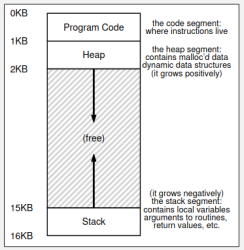
`x` going out of scope on deallocates only `x` itself (the box holding `int *`) – the memory allocated by `malloc` remains until freed by `free`

Address Space Layout

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  printf("location of code : %p\n", main);
  printf("location of heap : %p\n", malloc(100e6));
  int x = 3;
  printf("location of stack: %p\n", &x);
  return x;
}
```

When run on a 64-bit Mac, we get the following output:

```
location of code : 0x1095afe50
location of heap : 0x1096008c0
location of stack: 0x7fff691aea64
```



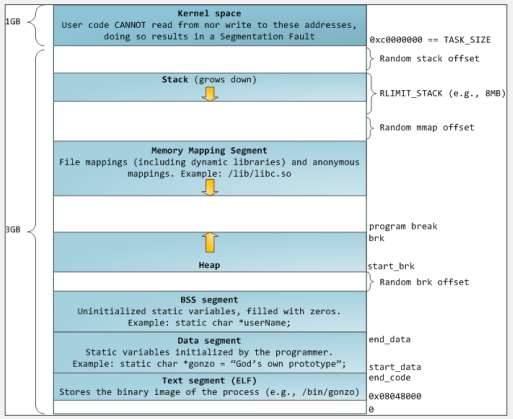
when run on x86_64 Linux with a v6.8 kernel –

```
location of code : 0x5951ce223189
location of heap : 0x7678132a1010
location of stack: 0x7fff72055ca4
```

Linux utilizes *Address Space Layout Randomization* (ASLR) – not having key code in predictable places makes it harder exploit memory corruption vulnerabilities

```
ssb@zhalen:~/mathcs/courses/331/s26/workspace/inclass/vm-intro$ ./va
location of code : 0x6248fbb47189
location of heap : 0x7e2458aa1010
location of stack: 0x7ffd1ac40dd4
ssb@zhalen:~/mathcs/courses/331/s26/workspace/inclass/vm-intro$ ./va
location of code : 0x620191eff189
location of heap : 0x76e1512a1010
location of stack: 0x7ffcdb8a0814
ssb@zhalen:~/mathcs/courses/331/s26/workspace/inclass/vm-intro$ ./va
location of code : 0x62dd4590c189
location of heap : 0x7d85a94a1010
location of stack: 0x7ffc064f2104
```


32-bit Linux Address Space Layout



malloc

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

- takes the size (in bytes) to allocate
  - use `sizeof(type)` to determine how many bytes are needed for one value of type *type* rather than hardcoding a number

```
double *d = (double *) malloc(sizeof(double));
```

  - technical note: `sizeof` is an operator rather than a function call – it is evaluated at compile time
    - you can pass in a variable instead of a type (e.g. `sizeof(x)`) but it will only report size info known at compile time – how big something of x's type is, not the size of a `malloc`ed block that `x` is pointing to
  - for strings, don't forget to allocate space for the null terminator
    - `char *s = malloc(strlen(src) + 1);` or
    - `char *s = malloc((strlen(src) + 1) * sizeof(char));`
  - for copying strings, prefer `strdup(c)`
- returns the (virtual) address of the beginning of a newly-allocated contiguous block of the specified size
  - cast to *type* *
  - returns `NULL` if the allocation fails

free

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

- deallocates the memory pointed to by `x`
  - memory allocation library tracks the size of the chunk of memory allocated

Other Flavors

```
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

- `calloc()`
  - zeroes the allocated memory before returning

> The **calloc**() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If nmemb or size is 0, then **calloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**(). If the multiplication of nmemb and size would result in integer overflow, then **calloc**() returns an error. By contrast, an

- `realloc()`
  - results in a (larger) chunk of memory with the same contents

> The **realloc**() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If ptr is NULL, then the call is equivalent to malloc(size), for all values of size; if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to **malloc**(), **calloc**(), or **realloc**(). If the area pointed to was moved, a free(ptr) is done.

- C provides direct access to memory through pointers
- data access and traversal rely on pointer arithmetic
- accessing an illegal address triggers a segmentation fault (segfault) but there is no routine bounds-checking for arrays or blocks of allocated memory

---

Safety depends on careful reasoning and disciplined programming – the language/runtime does not enforce it.

- just because it compiles doesn't mean it is correct
  - program can try to read from or write to an invalid memory address at runtime
- just because it runs once doesn't mean it is correct or that it will run the next time
  - e.g. buffer overflow has the potential to overwrite other things in memory – but there may not be anything important stored there

Common Errors

Things causing crashes or (worse) incorrect behavior –

- forgetting to allocate memory
  - many string routines (e.g. `strcpy`) expect that the necessary memory has already been allocated
    - `strdup` is an exception – it allocates the memory itself, though the programmer must still free
- not allocating enough memory (buffer overflow)
- forgetting to initialize allocated memory
  - `malloc` allocates memory but does not initialize it
- freeing memory before you are done with it (dangling pointer)
- freeing memory repeatedly (double free)
  - the consequences are undefined; crashing is a common occurrence
- calling `free()` incorrectly (invalid free)

Common Errors

Generally less immediately consequential, but still poor practice –

- forgetting to free memory (memory leak)
  - for short-lived programs, the program will generally still run correctly
    - all memory allocated to a process is deallocated when the process ends
  - in long-running programs, memory leaks can eventually result in running out of memory

Address Translation