University of Glasgow

Resource Ownership and Memory Management

Advanced Systems Programming (H)

Lecture 5

Outline

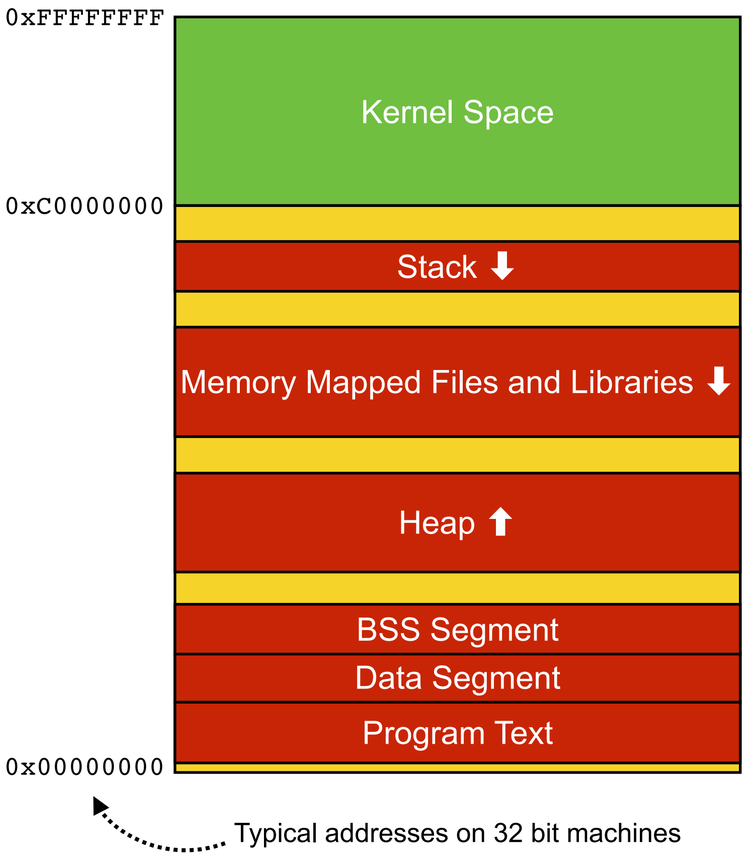
- Memory
  - How is a process stored in memory?
  - What memory has to be managed?
- Memory management
  - Reference counting
  - Region-based memory management
- Resource management

Memory

- How is a process stored in memory?
- What memory has to be managed?

Layout of a Processes in Memory

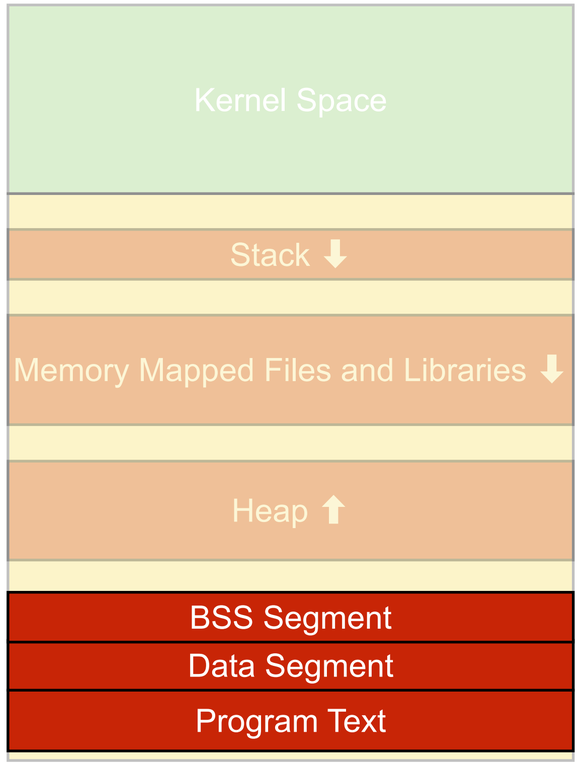
- To understand memory management, must understand what memory is to be managed:
  - Program text, data, and global variables
  - Heap allocated memory
  - Stack
  - Memory mapped files and shared libraries
  - Operating system kernel



See also:
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

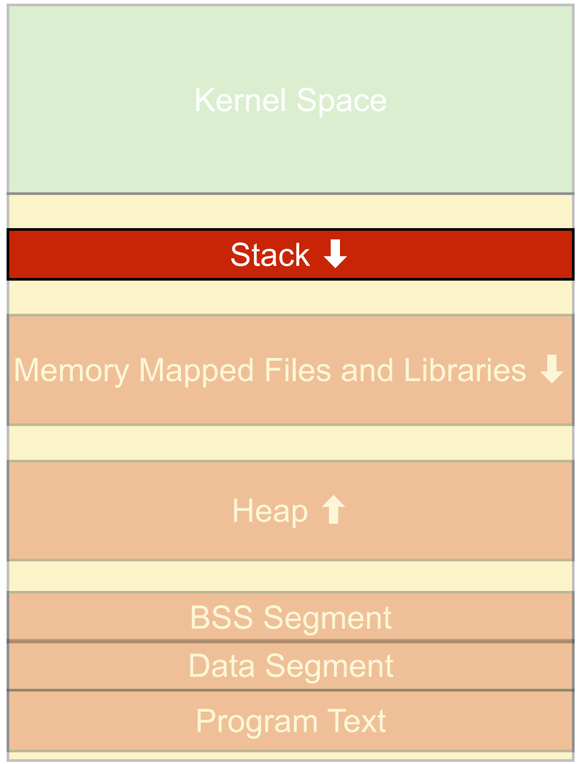
Program Text, Data, and BSS

- Program text, data, global variables occupy lowest address space
  - Page with address zero reserved to trap null-pointer dereferences
  - **Program Text** is compiled machine code of program
  - **Data segment** is variables initialised in source code
    - String literals, initialised **`static`** global variables in C
    - Known at compile time, loaded along with program text
  - **BSS segment** reserved for uninitialised **`static`** global variables
    - “block started by symbol” – name is historical relic
    - Initialised to zero by runtime when the program loads



The Stack

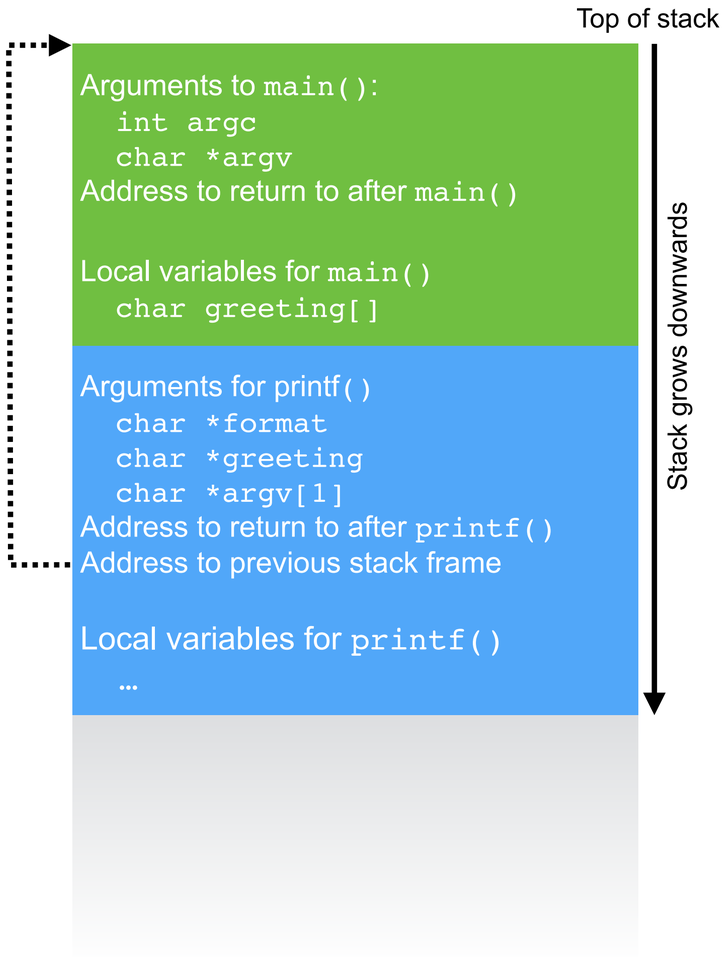
- **The stack** holds function parameters, return address, local variables
  - Function calls push data onto stack, growing down
    - Parameters for the function; return address; pointer to previous stack frame; local variables
    - Data removed, stack shrinks, when function returns – stack managed automatically
      - Compiler generates code to manage the stack as part of the compiled program
      - The calling convention for functions, how parameters are pushed onto the stack, standardised for given processor and programming language
    - The operating system generates the stack frame for **`main()`** when the program starts
- Ownership of stack memory follows function invocation



Function Calling Conventions

- Example: code and contents of stack while calling **`printf()`** in code below:

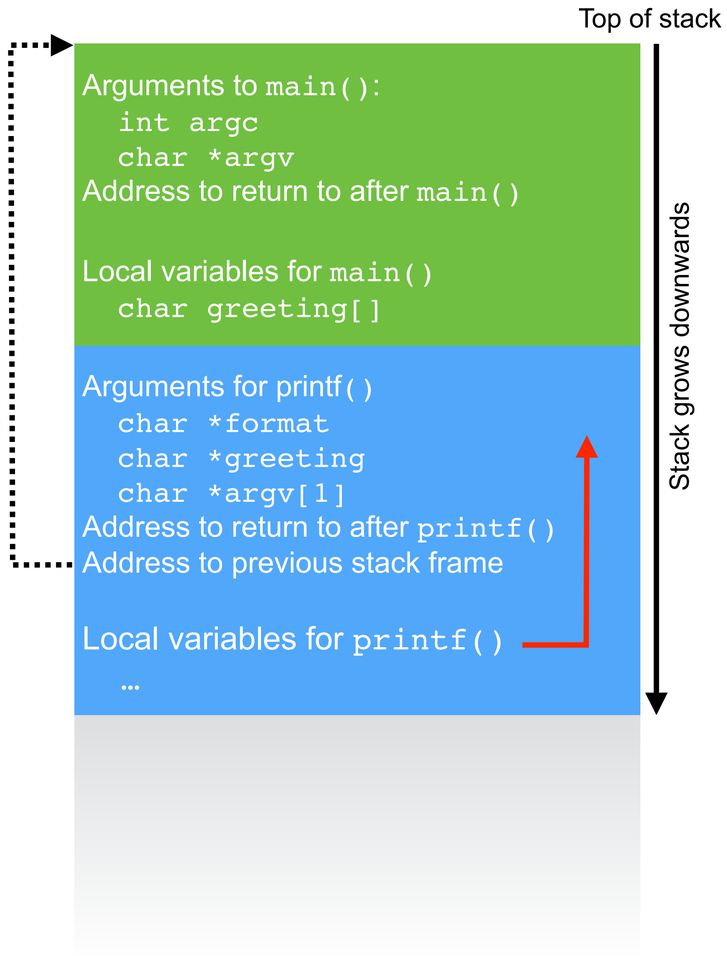
```
#include <stdio.h>

int
main(int argc, char *argv[])
{
  char greeting[] = “Hello”;

  if (argc == 2) {
    printf(“%s, %s\n”, greeting, argv[1]);
    return 0;
  } else {
    printf(“usage: %s <name>\n”, argv[0]);
    return 1;
  }
}
```



- Address of the previous stack frame is stored for ease of debugging, so stack trace can be printed, so it can easily be restored when function returns

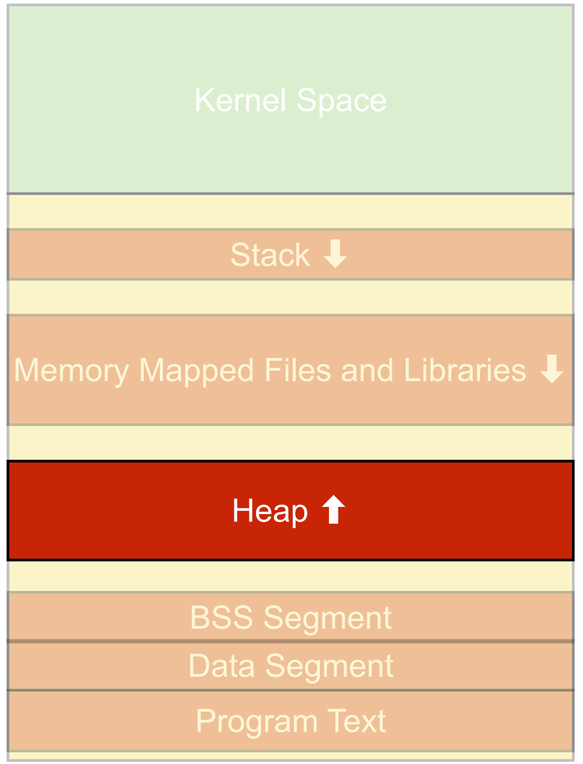
Buffer Overflow Attacks

- Classic buffer overflow attack:
  - Language not type safe, doesn't enforce abstractions
  - Write past array bounds → overflows space allocated to local variables, overwrites return address and following data
  - Contents valid machine code; the overwritten function return address is made to point to that code
  - When function returns, code written during overflow is executed
  - http://phrack.org/issues/49/14.html#article
- Workarounds:
  - Marks stack as non-executable
  - Randomise top of stack address each program run
  - Various more complex buffer overflow attacks still possible; e.g., see "return-oriented programming"
- Solution: use a language that is type safe and enforces array bounds checks



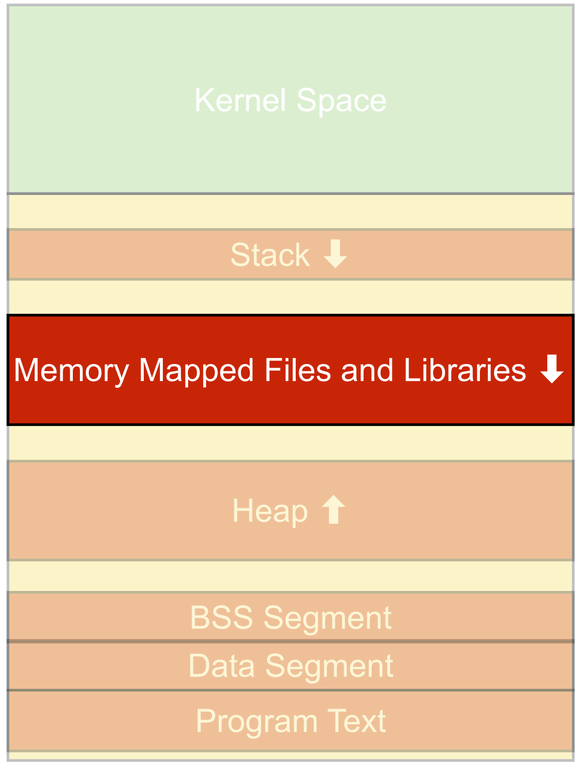
The Heap

- **The heap** holds explicitly allocated memory
  - Allocated using **`malloc()`**/**`calloc()`** in C
  - Starts at a low address in memory; later allocations follow in consecutive addresses
    - Sometimes padded to align to a 32 or 64 bit boundary, depending on processor
    - Modern **`malloc()`** implementations are thread aware, split heap into different parts different threads to avoid cache sharing
  - Memory management primarily concerned with reclaiming heap memory
    - Manually, using **`free()`**
    - Automatically via reference counting/garbage collection
    - Automatically based on regions and lifetime analysis



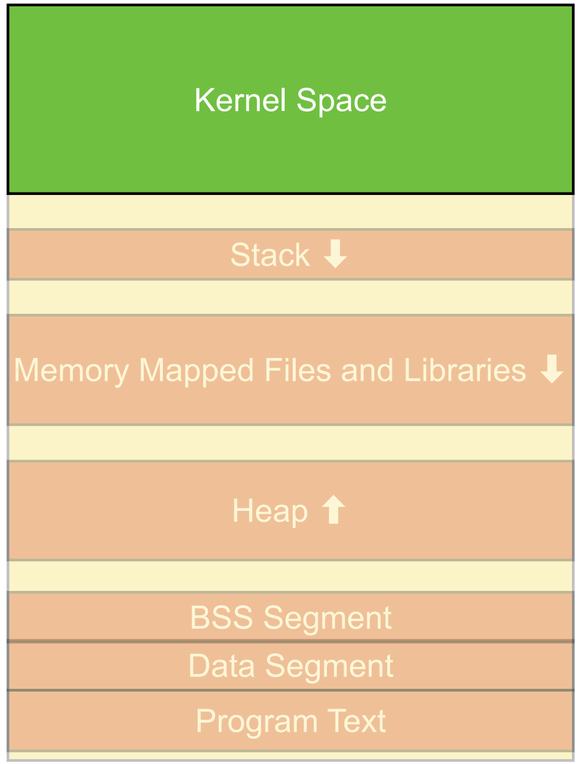
Memory Mapped Files and Shared Libraries

- **Memory mapped files** allow data on disk to be directly mapped into address space
  - Mappings created using **`mmap()`** system call
    - Returns a pointer to a memory address that acts as a proxy for the start of the file
    - Reads from/writes to subsequent addresses acts on the underlying file
    - File is demand paged from/to disk as needed – only the parts of the file that are accessed are read into memory (granularity depends on virtual memory system; often 4k pages)
    - Useful for random access to parts of files
- Used to map **shared libraries** into memory
  - **`.so`** files on Unix, **`.dll`** files on Windows



The Kernel

- Operating system **kernel** resides at top of the address space
  - Not directly accessible to user-space programs
    - Attempt to access kernel → segmentation violation
    - The **`syscall`** instruction in x86_64 assembler calls into the kernel after permission check
  - Kernel can read/write memory of user processes



Memory

- How is a process stored in memory?
- What memory has to be managed?