University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Memory Management

Advanced Operating System

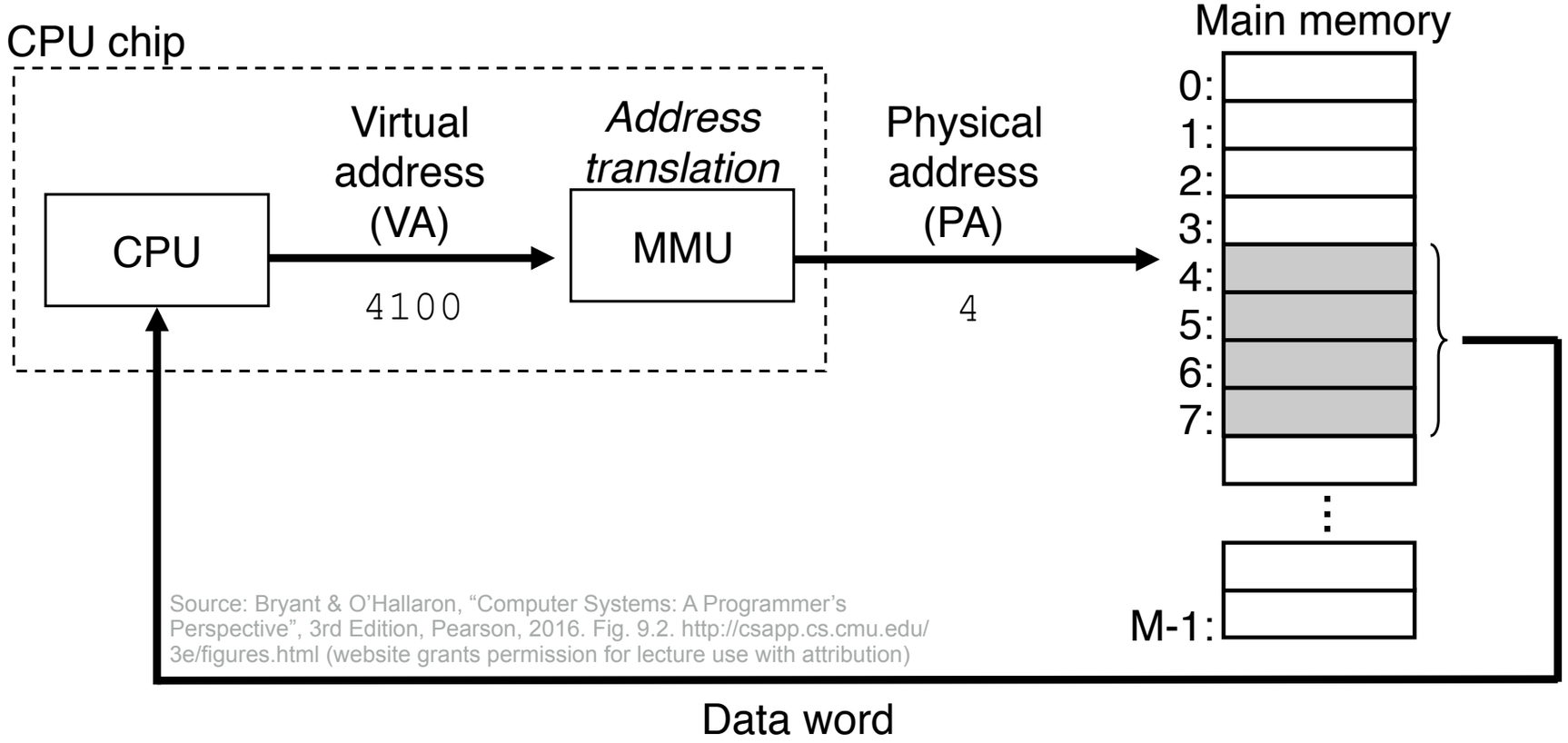
Lecture 5

Lecture Outline

- Virtual memory
- Layout of a processes address space
  - Stack
  - Heap
  - Program text, shared libraries, etc.
- Memory management

Virtual Memory (1)

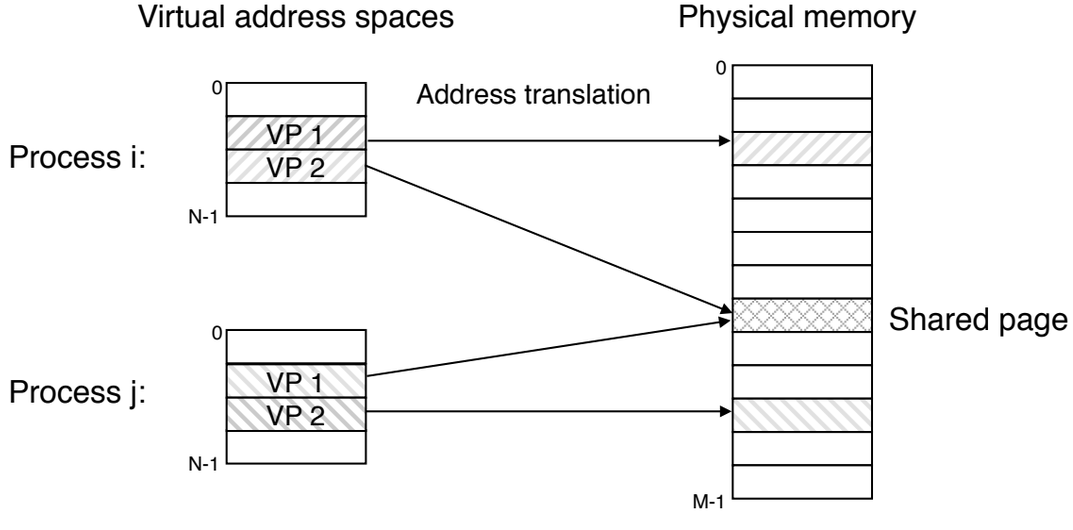
- Processes see *virtual addresses* – each process believes it has access to the entire address space
- Kernel programs the MMU (“memory management unit”) to translate these into physical addresses, representing real memory – mapping changed each context switch to another process



Source: Bryant & O’Hallaron, “Computer Systems: A Programmer’s Perspective”, 3rd Edition, Pearson, 2016. Fig. 9.2. http://csapp.cs.cmu.edu/3e/figures.html (website grants permission for lecture use with attribution)

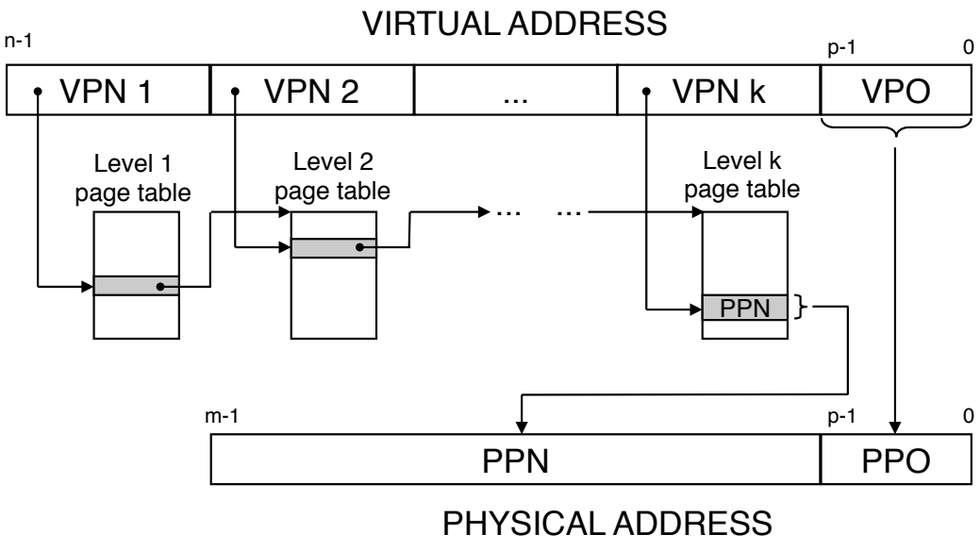
Virtual Memory (2)

- Virtual-to-physical memory mapping can be unique or shared
  - Unique mappings typical – physical memory owned by a single process
  - Shared mappings allow one read-only copy of a shared library to be accessed by many processes
  - Memory protection done at page level – each page can be readable, writable, executable…
- Mapping is on granularity of pages



Source: Bryant & O'Hallaron, "Computer Systems: A Programmer's Perspective", 3rd Edition, Pearson, 2016. Fig. 9.9. http://csapp.cs.cmu.edu/3e/figures.html (website grants permission for lecture use with attribution)

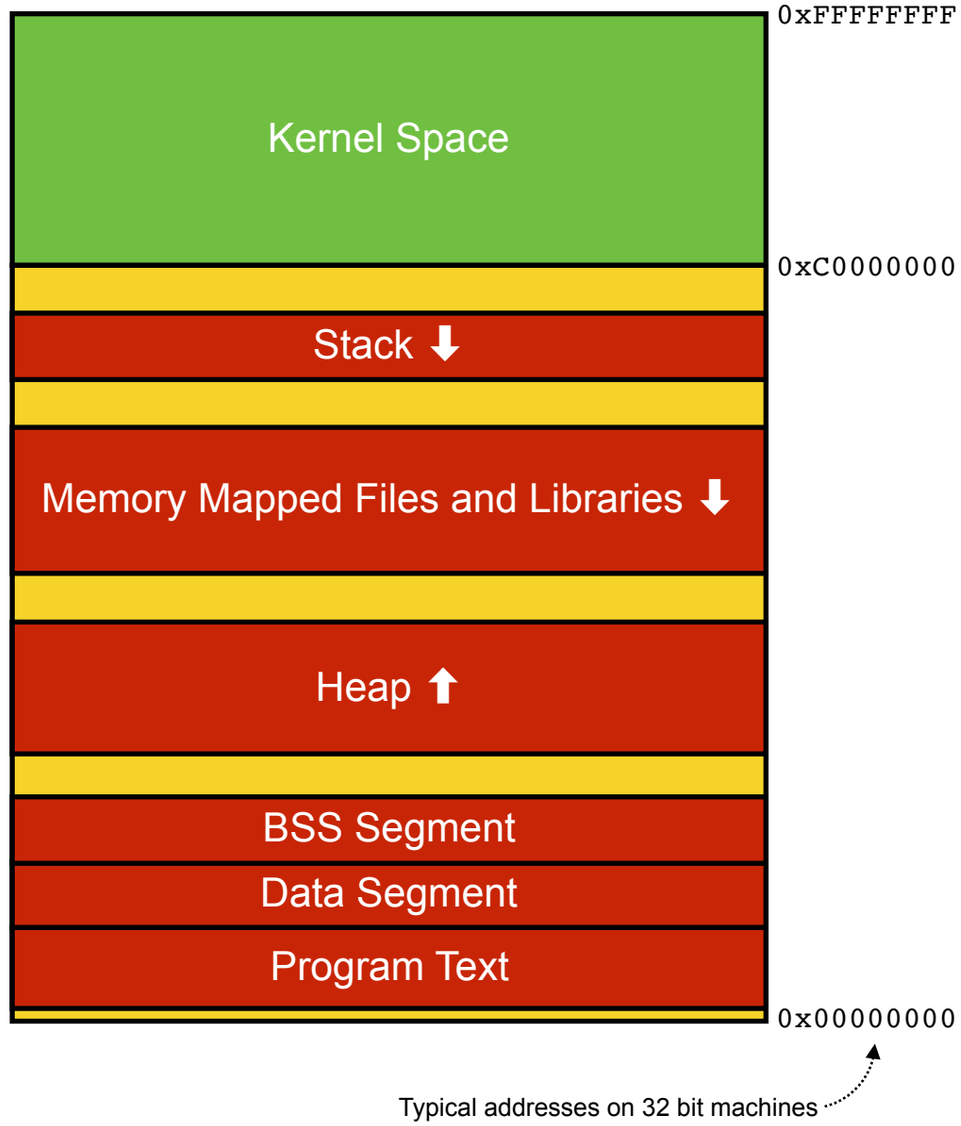
- Virtual to physical mapping typically a multi-level process
  - Page tables organised as a tree; only entries that are populated, and their ancestors, exist to save space
  - Translation managed by the kernel – invisible to applications



Source: Bryant & O'Hallaron, "Computer Systems: A Programmer's Perspective", 3rd Edition, Pearson, 2016. Fig. 9.18. http://csapp.cs.cmu.edu/3e/figures.html (website grants permission for lecture use with attribution)

Layout of a Processes Address Space

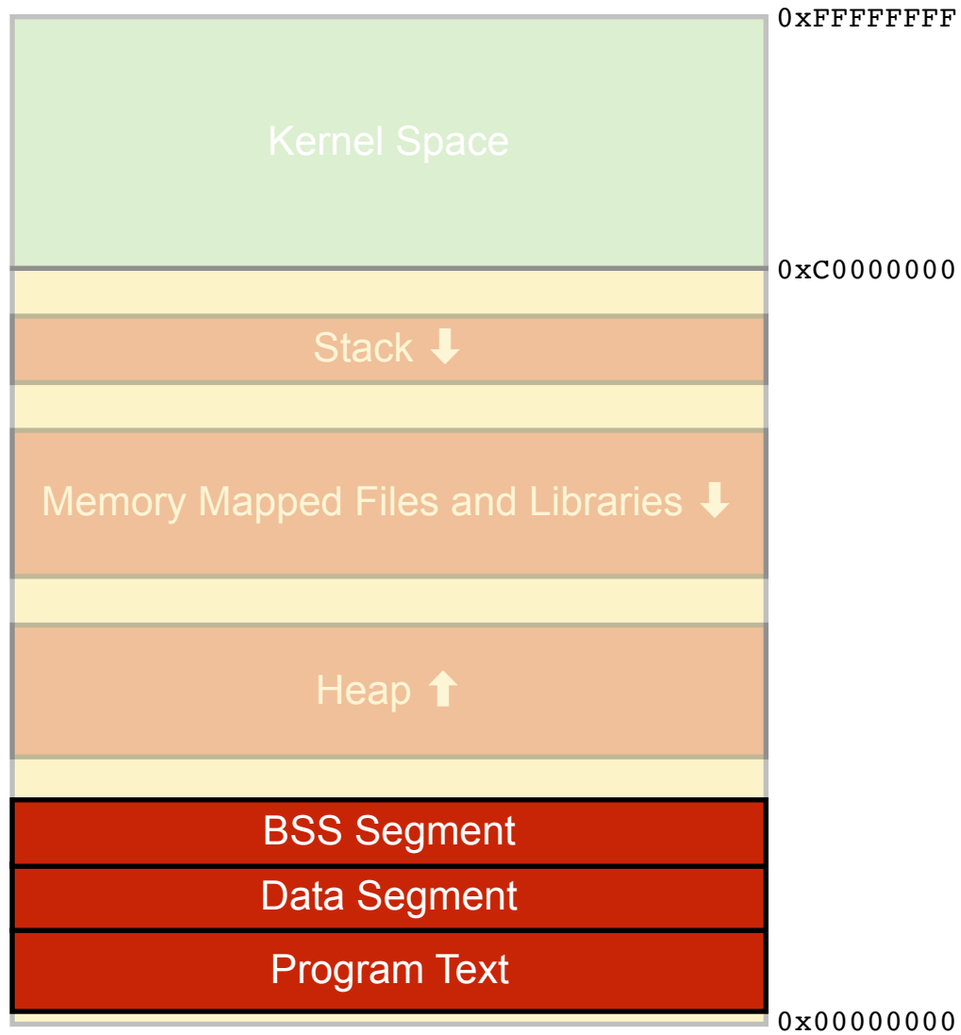
- Typical layout of process address space, in virtual memory
  - Program text, data, and global variables at bottom of address space; heap follows, growing upwards
  - Kernel at top of address space; stack grows down, below kernel
  - Memory mapped files and shared libraries between these



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

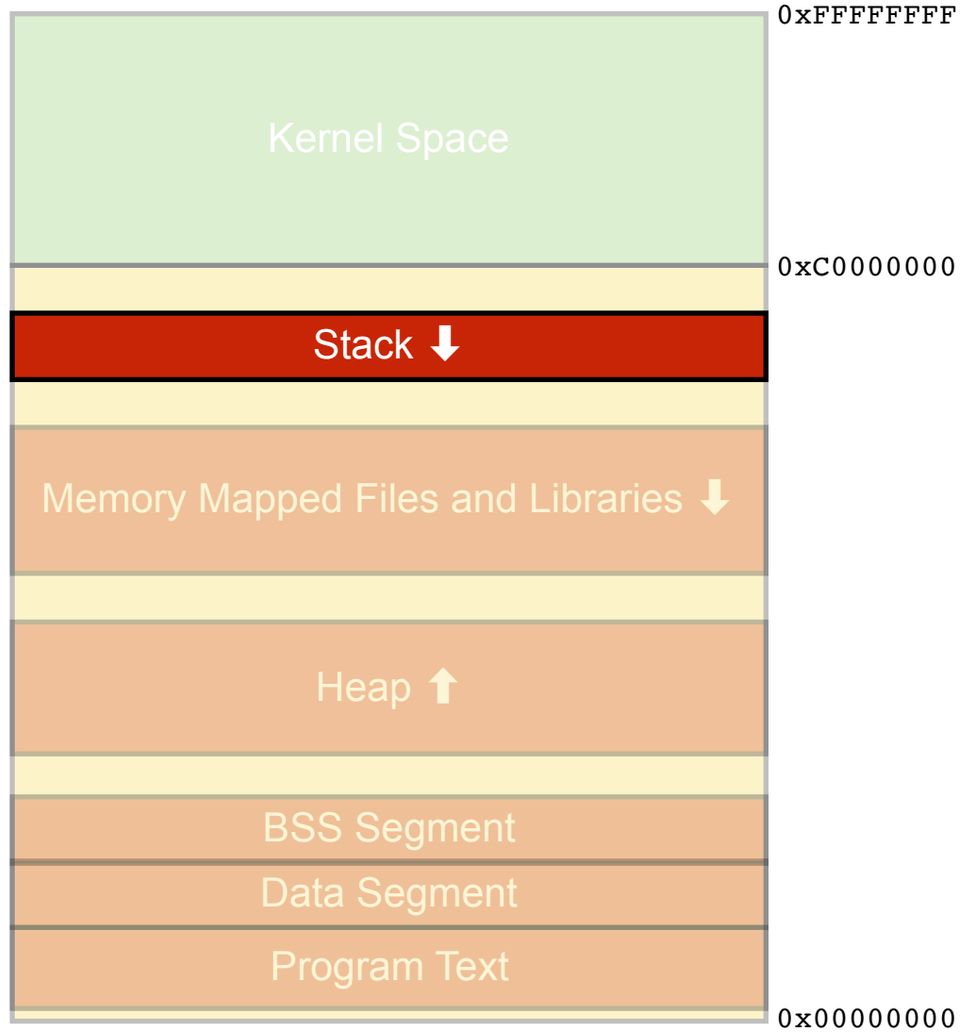
Program Text, Data, and BSS

- Compiled machine code of the program text typically occupies bottom of address space
  - Lowest few pages above address zero reserved to trap null-pointer dereferences – access will trigger kernel trap, killing process with segmentation fault
  - Program text is marked read-only
- Data segment contains variables initialised in the source code
  - E.g., `static char *hello = “Hello, world!”;` at the top level of a C program would allocate space in the data segment
  - Data segment is known at compile time, loaded along with program text
- BSS segment holds space for uninitialised global variables
  - BSS is “block started by symbol”; name is historical relic
  - Initialised to zero by the runtime when the program loads (C standard requires this for uninitialised static variables)



The Stack

- Stack holds function parameters, return address, and local variables
  - Starts at high address in memory
  - Each function called pushes data onto stack, growing stack downwards towards lower memory addresses
    - Parameters for the function; return address; pointer to previous stack frame; local variables
  - When the function returns, that data is removed from the stack, which shrinks
- The stack is managed automatically
  - The operating system generates the stack frame for `main()` when the program starts
  - The compiler generates the code to manage the stack when it compiles the program text
  - The calling convention for functions – how parameters are pushed onto the stack – is standardised for a given processor and programming language
- Ownership tracks function execution



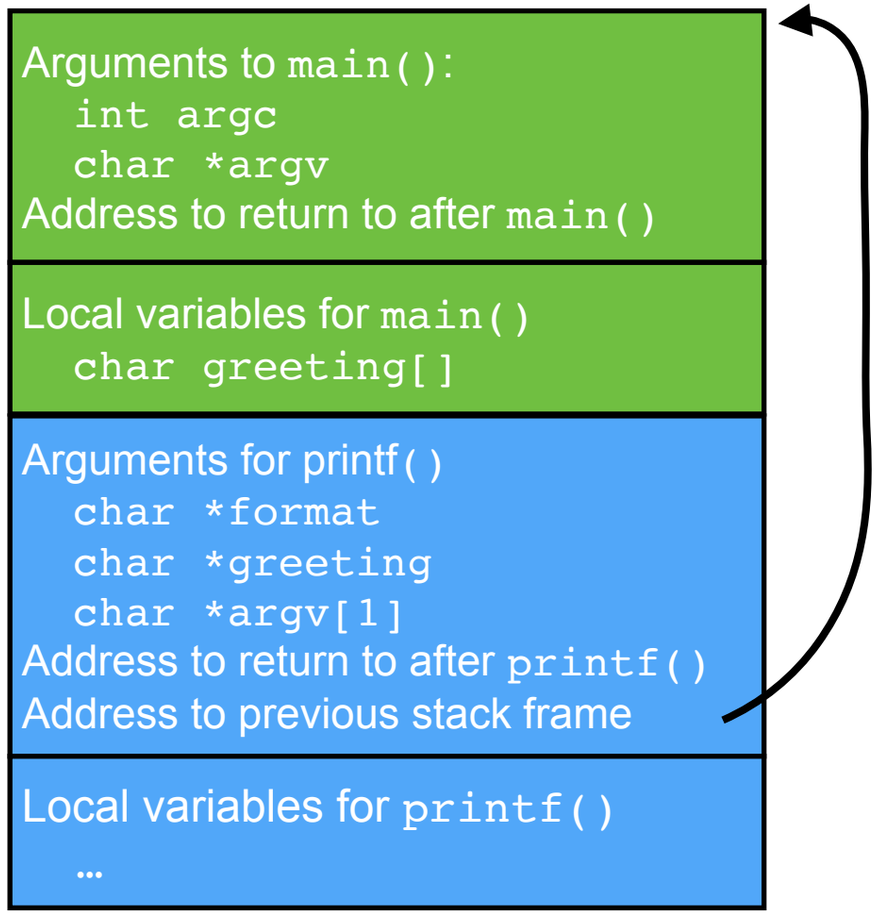
Function Calling Conventions

- Example: code and contents of stack while calling printf() function

```
#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;
  }
}
```

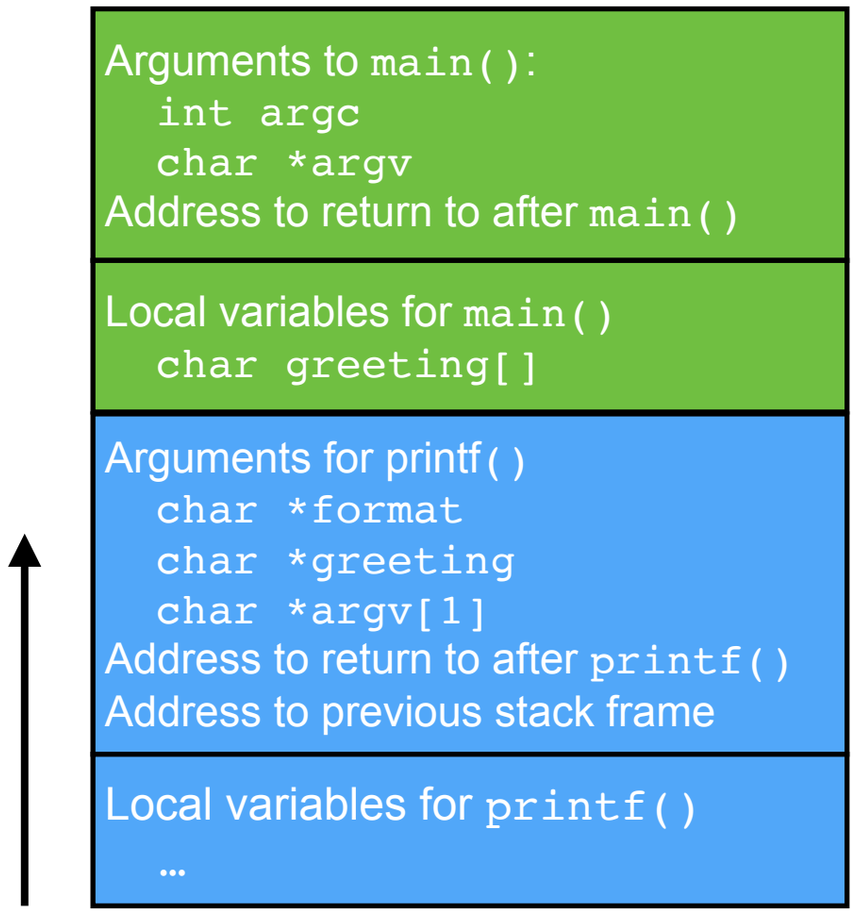


- The address of the previous stack frame is stored for ease of debugging, so stack trace can be printed, and so it can easily be restored when function returns

Buffer Overflow Attacks

- Classic buffer overflow attack:
  - Language runtime doesn't check array bounds
  - Writing too much data overflows the space allocated to local variables, overwrites function return address, and following data
  - Contents are valid machine code; the overwritten function return address is made to point to that code
  - When function returns, code written during the overflow is executed

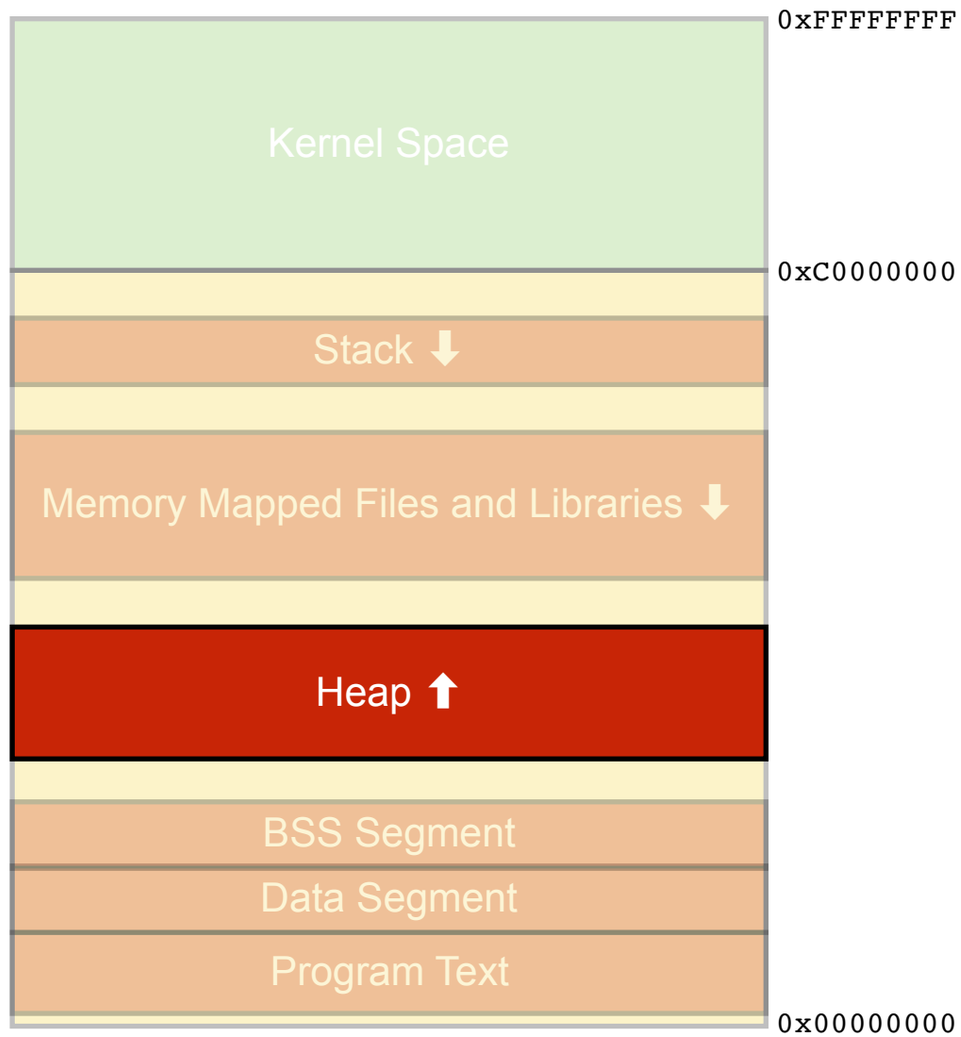
- Solve by marking stack as non-executable, or randomising start address of stack each time program runs
  - Or use a language that enforces array bounds checks
  - Various more complex buffer overflow attacks still possible – e.g., see "return-oriented programming"



Local variables stored on stack immediately preceding function return address – overflowing local variable space will overwrite return address

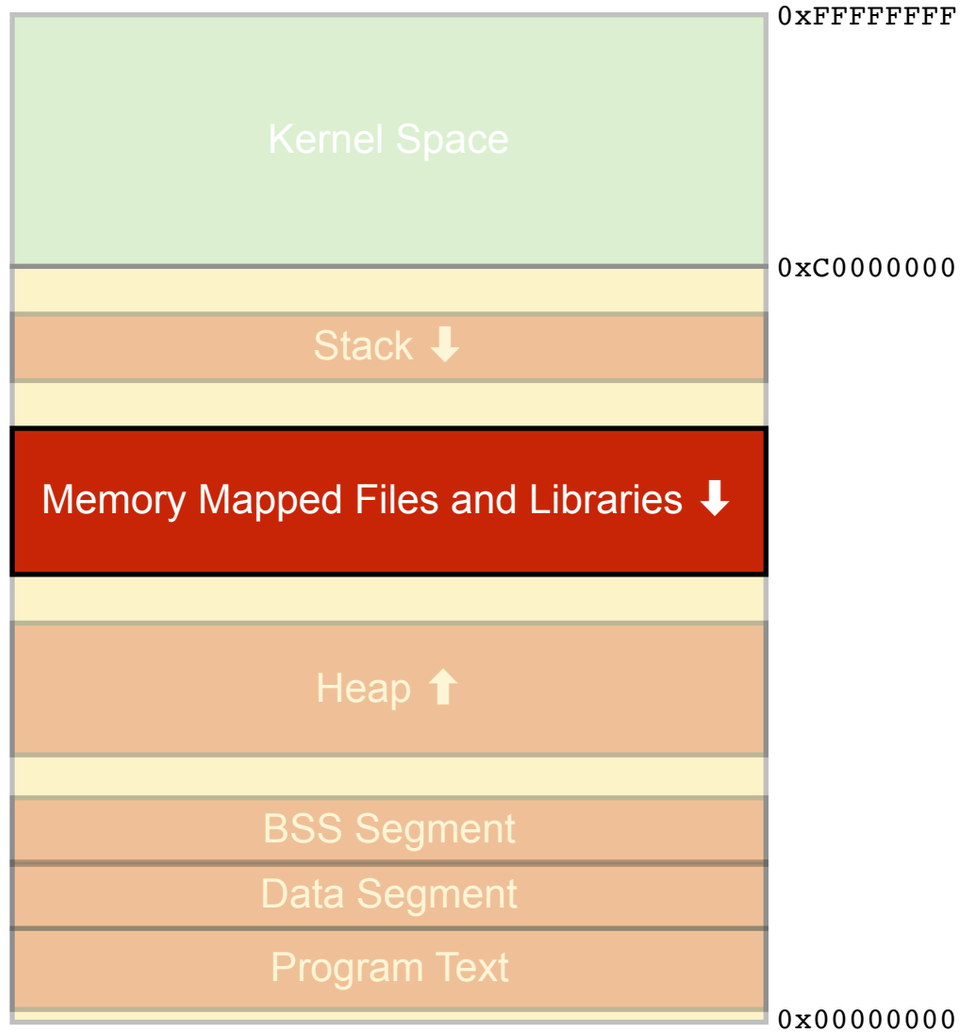
The Heap

- Explicitly allocated memory located in the heap
  - In C, memory allocated using `malloc()`/`calloc()`
  - In Java, objects allocated using `new`
  - …
- Starts at a low address in memory
- Consecutive allocations placed at increasing addresses in memory
  - Usually consecutive addresses, although some types of processor require allocations to be aligned to a 32 or 64 bit boundary
  - Modern `malloc()` implementations typically thread aware, and can use different regions of the heap for different threads, to avoid cache sharing
  - NUMA systems complicate heap allocation
- Memory management primarily concerned with managing the heap



Memory Mapped Files

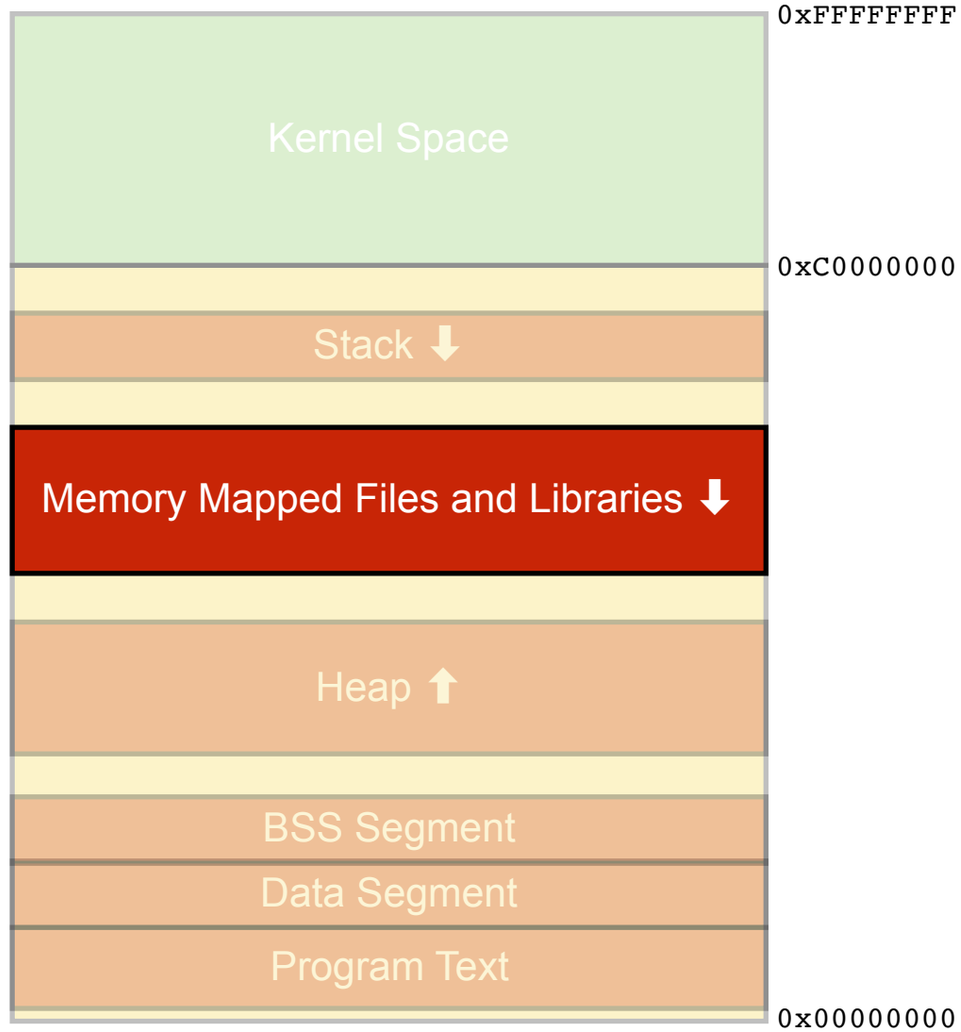
- The `mmap()` system call manipulates virtual memory mappings
- Common use: allows files to be mapped into virtual memory
  - Returns a pointer to a memory address that acts as a proxy for the start of the file
  - Reads from/writes to subsequent addresses acts on the underlying file
  - File is demand 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
- Can also be used to establish mappings for new virtual address space – i.e., to allocate memory
  - In modern Unix-like systems, this is how `malloc()` gets memory from the kernel



Shared Libraries

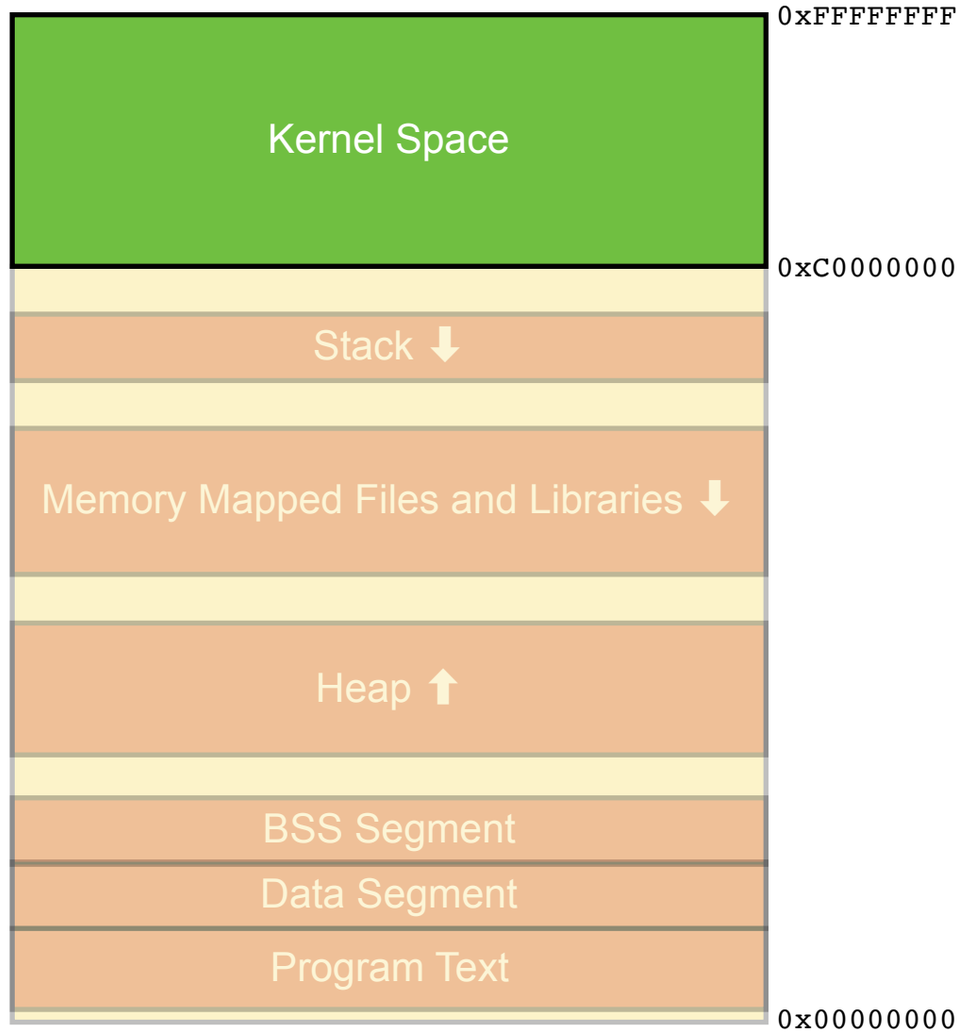
- Code for shared libraries also mapped into memory between heap and stack
- Virtual memory system enforces these are read-only

- One copy only in physical memory; virtual address can differ for each process



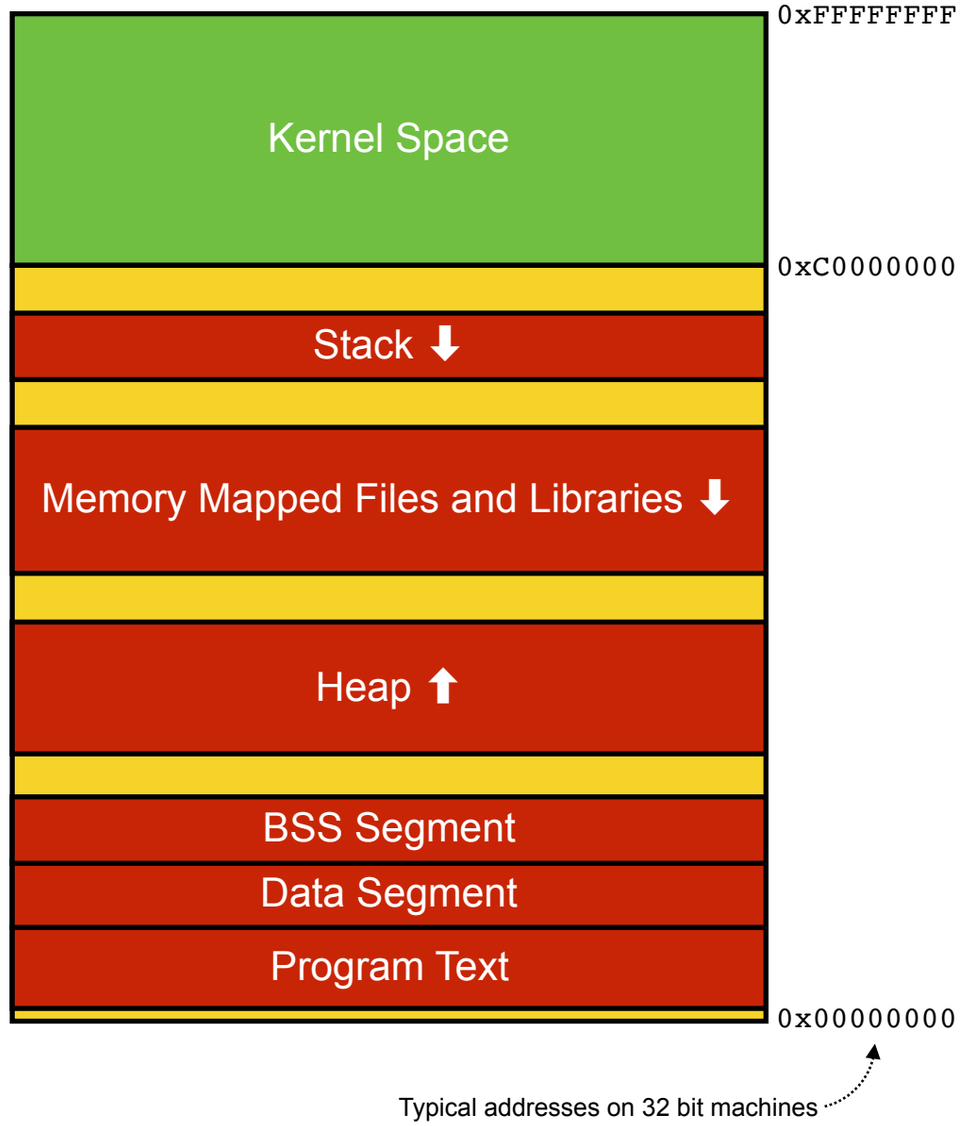
The Kernel

- Operating system kernel owns top part of the address space
  - Not accessible to user-space programs – attempting to read kernel memory gives segmentation violation
  - The `syscall` instruction in x86_64 assembler jumps to the kernel, to one of a set of pre-defined system calls – switches processor to privileged mode, reconfigures the virtual memory for kernel mode
  - Kernel can read/write memory of user processes



Address Space Layout Randomisation

- Relative layout of memory regions generally fixed – always in the same order
- Exact locations in memory randomised on modern operating systems
  - Address of start of stack
  - Address of start of heap
  - Addresses where shared libraries are loaded
  - Addresses representing memory mapped files
- Complicates various types of buffer overflow attack if the addresses are unpredictable



Memory Management

- Stack memory allocated/reclaimed automatically
- Heap memory needs to be explicitly managed
  - Memory must be obtained from the kernel, managed by the application, and freed after use
  - Manually using, e.g., `malloc()` and `free()`
    - Implementation historically used `sbrk()` system call to increase the size of the heap as needed; modern systems use `mmap()` to establish new mapping for anonymous memory
    - The `malloc()` implementation sub-divides the space allocated by the kernel, using an internal persistent data structure to track what parts of the space are in use
    - Fragmentation after `free()` can be an issue – implementations often sub-divide the space, with different regions for allocations in different size ranges, to try to address this
    - Multi-threading can be an issue – implementations often sub-divide the space, with different threads allocating from different regions, to avoid cache contention
    - See jemalloc.net for a modern `malloc()` implementation with good documentation
  - Automatically, via region inference, reference counting, or using garbage collection – see lectures 6-8