University of Glasgow

School of Computing Science

Memory Management

Advanced Operating Systems (M)

Lecture 3

Colin Perkins | https://csperkins.org/ | Copyright © 2018 | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Lecture Outline

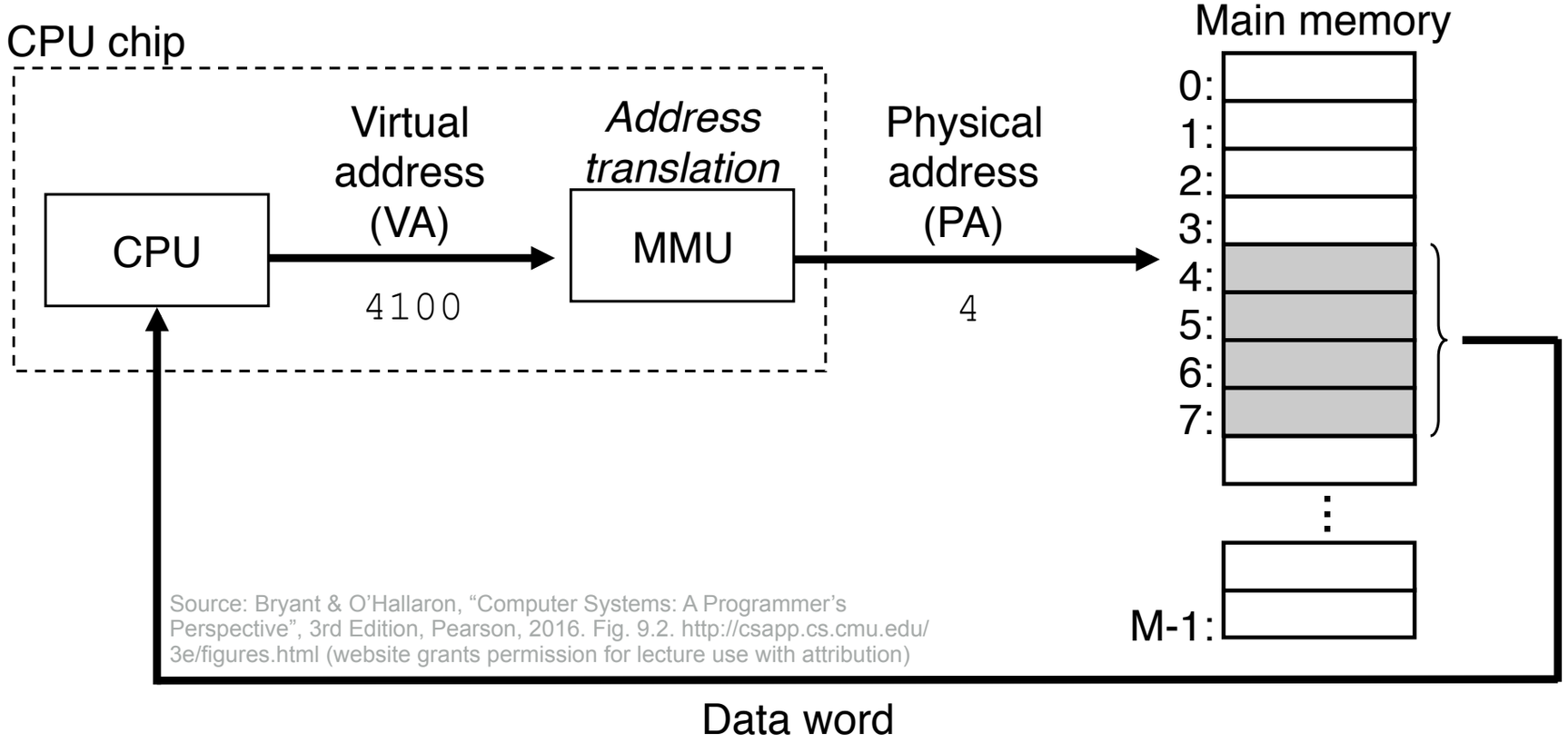
- Virtual memory
- Layout of a processes address space
  - Stack
  - Heap
  - Program text, shared libraries, etc.
- Memory management
  - Region-based Memory Management

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Virtual Memory and Process Address Space

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

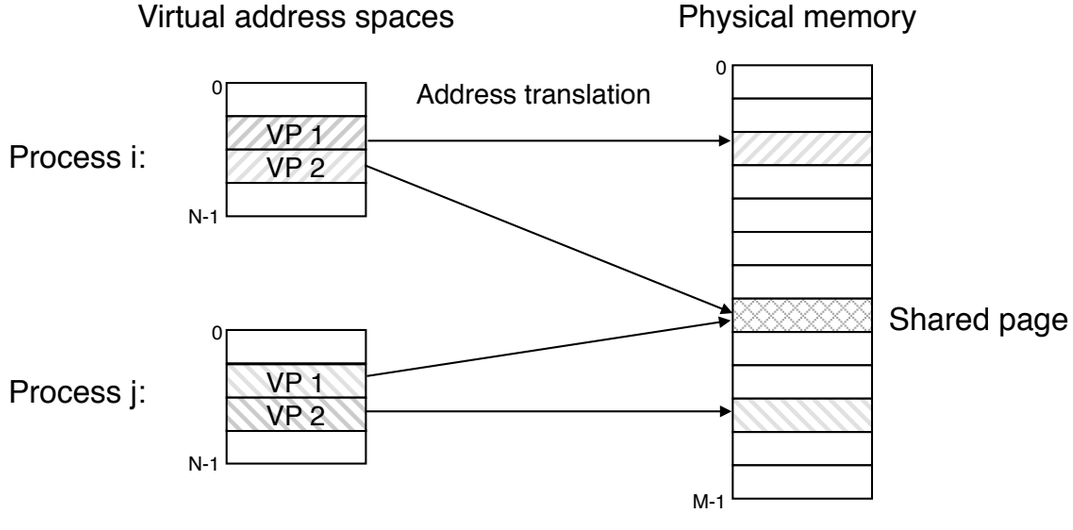


Source: Bryant & O’Hallaron, “Computer Systems: A Programmer’s Perspective”, 3rd Edition, Pearson, 2016. Fig. 9.2. http://csapp.cs.cmu.edu/3e/figures.html (website grants permission for lecture use with attribution)

CC BY ND

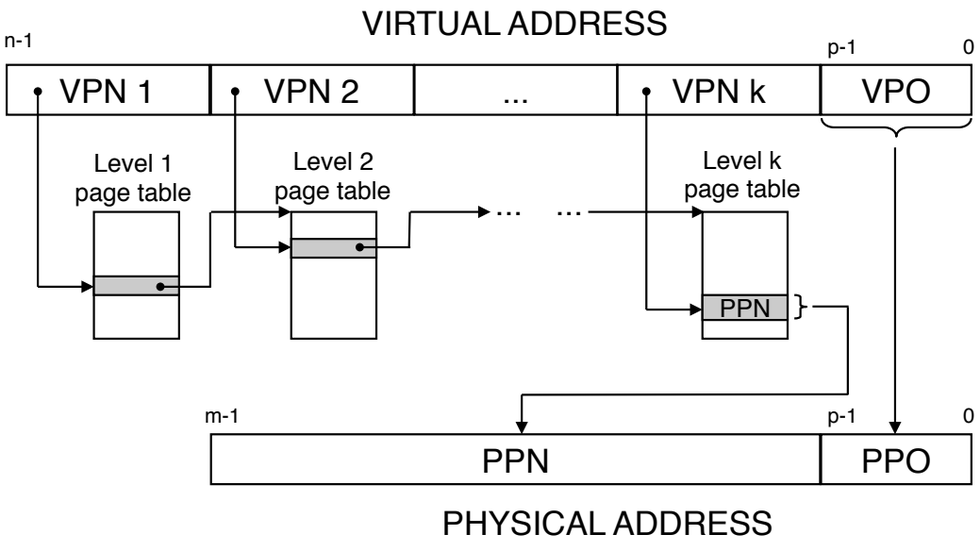
Virtual Memory (2)

- 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

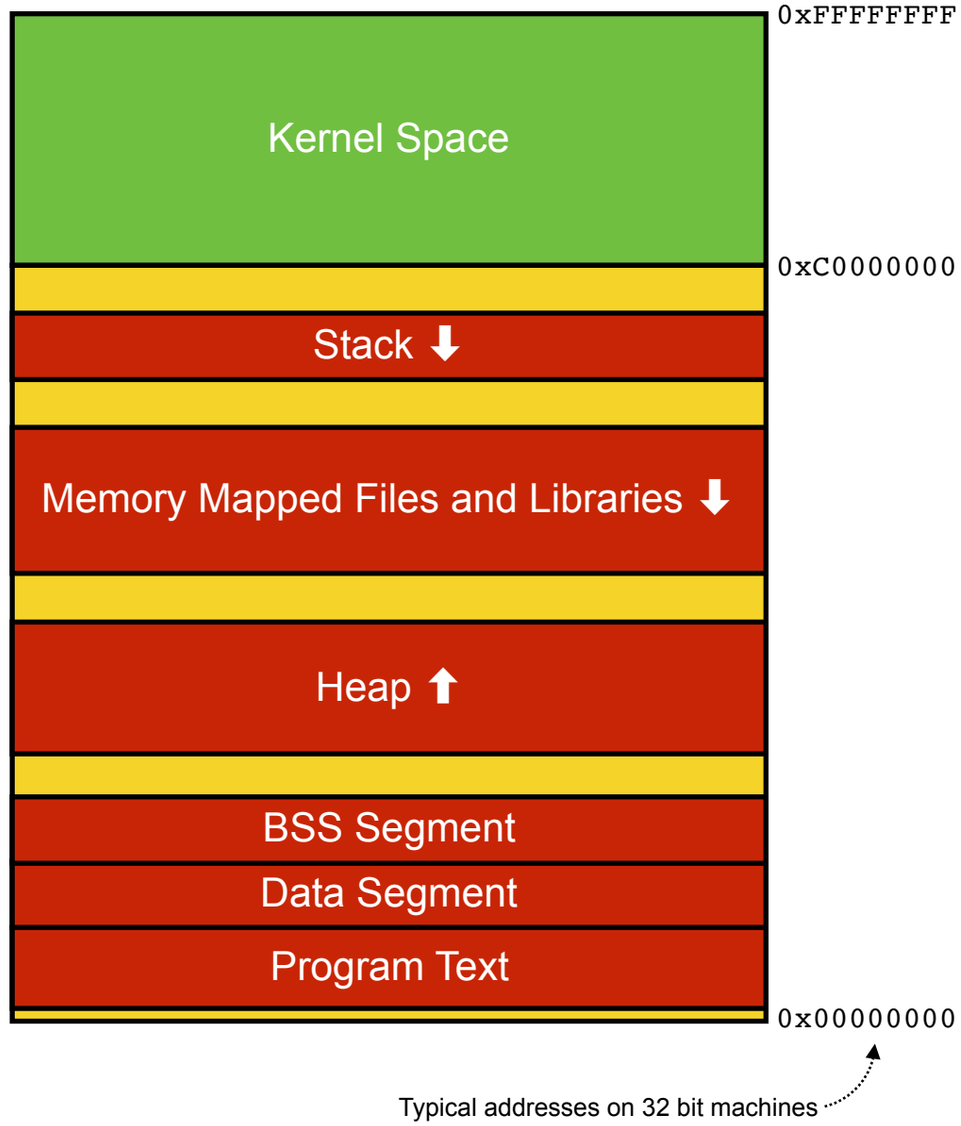


Source: Bryant & O'Hallaron, "Computer Systems: A Programmer's Perspective", 3rd Edition, Pearson, 2016. Fig. 9.18. http://csapp.cs.cmu.edu/3e/figures.html (website grants permission for lecture use with attribution)

CC BY ND

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

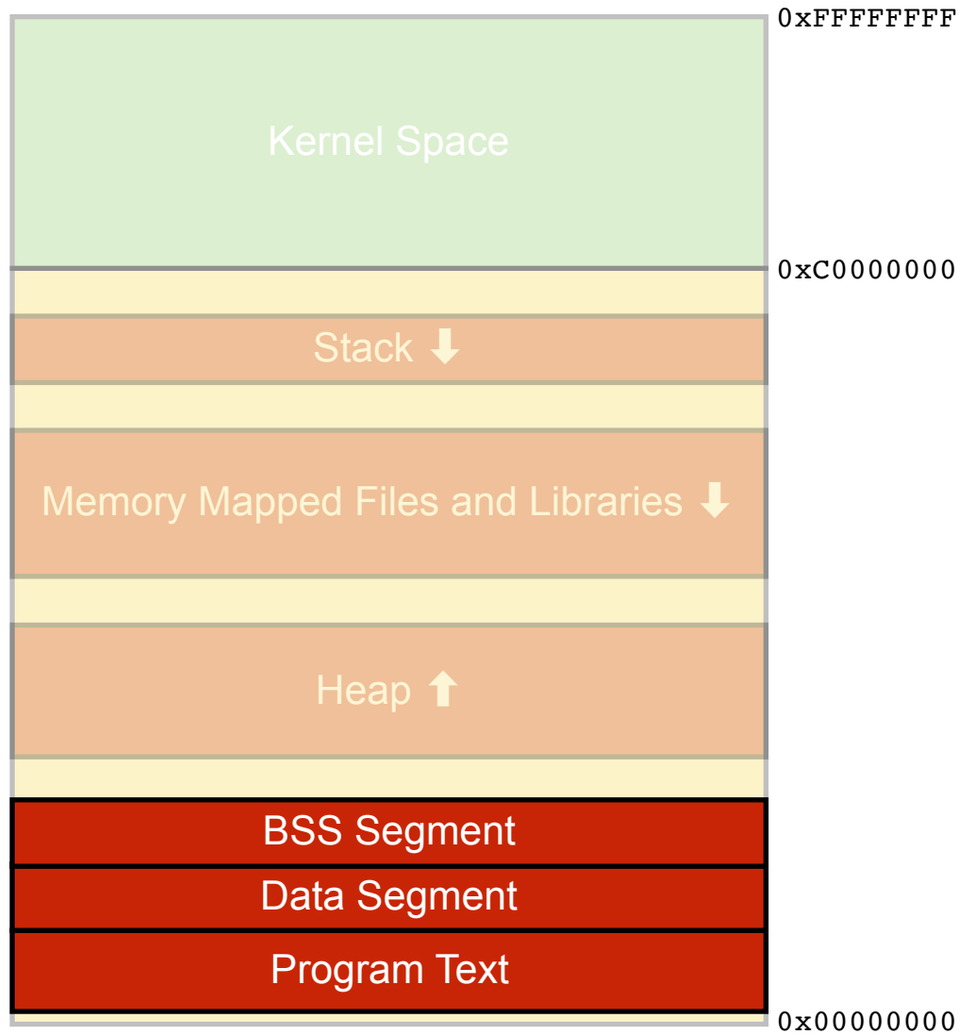


See also http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

CC BY ND

Program Text, Data, and BSS

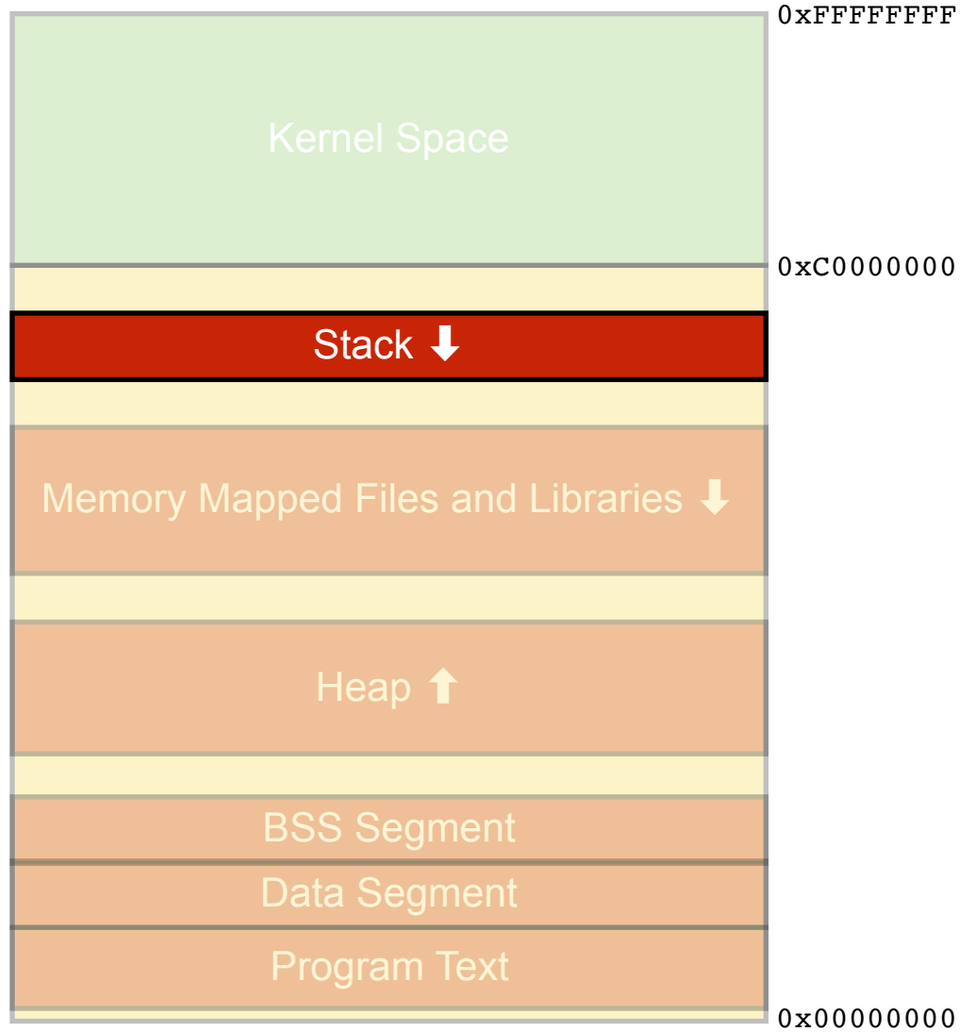
- Compiled machine code of the program text typically occupies bottom of address space
  - Lowest few pages above address zero reserved to trap null-pointer dereferences – access will trigger kernel trap, killing process with segmentation fault
  - Program text is marked read-only
- Data segment contains 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)



CC BY ND

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



CC BY ND

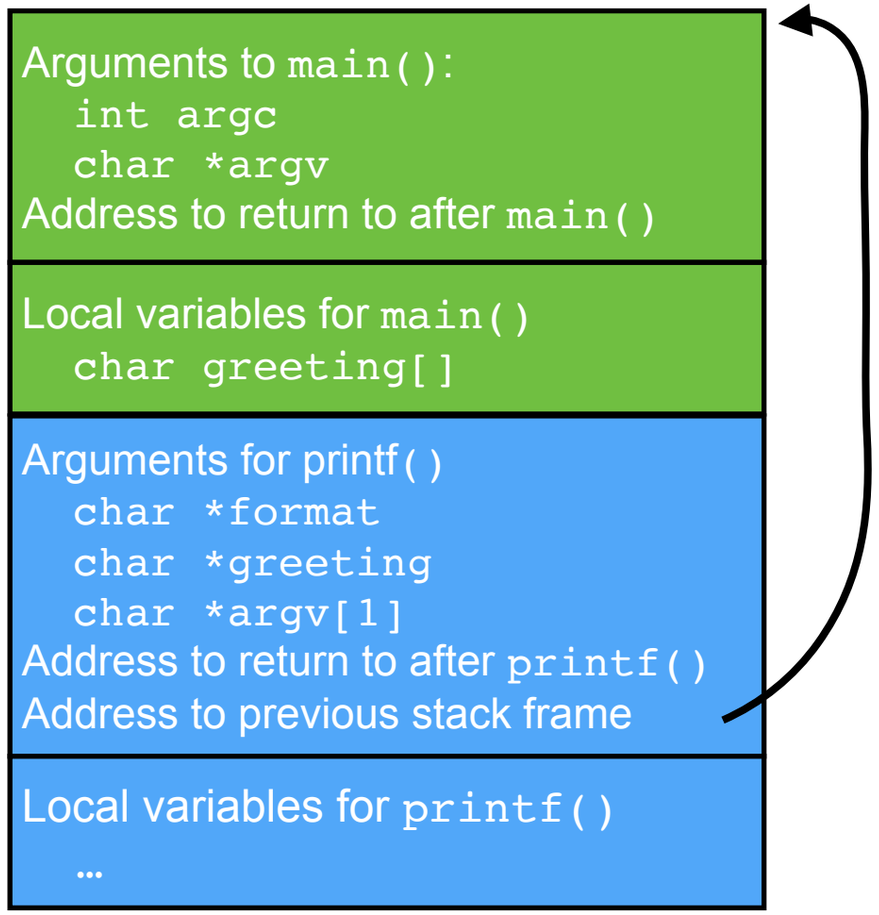
Function Calling Conventions

- 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;
  }
}
```



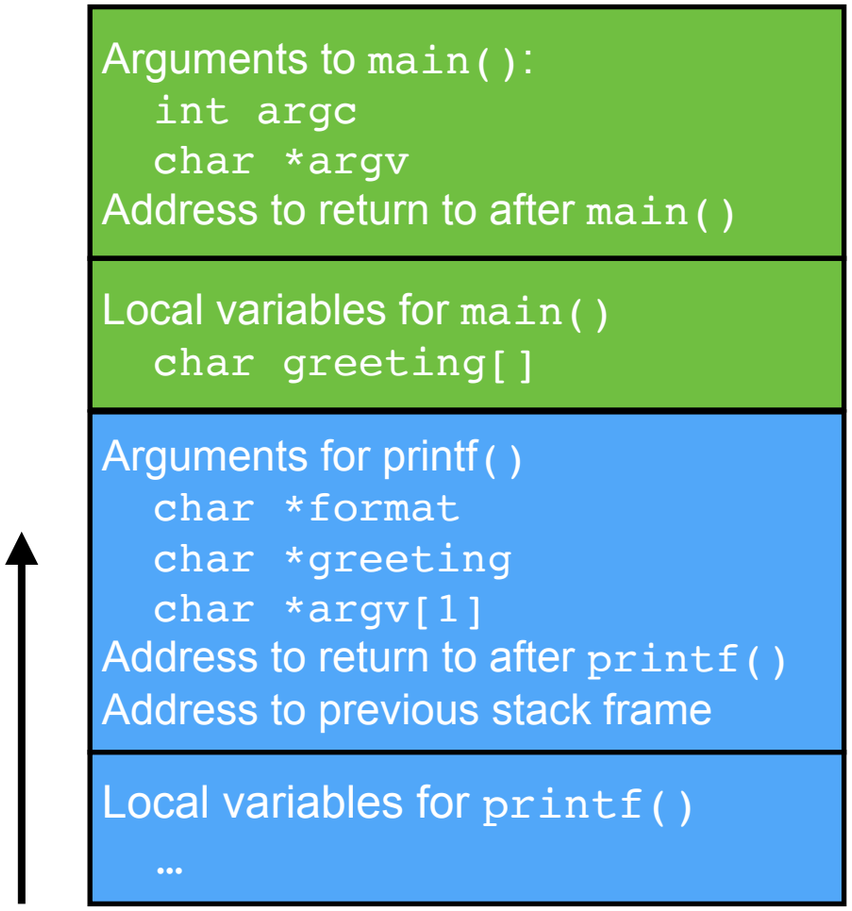
- Address of the previous stack frame is stored for ease of debugging, so stack trace can be printed, so it can easily be restored when function returns

CC BY ND

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"

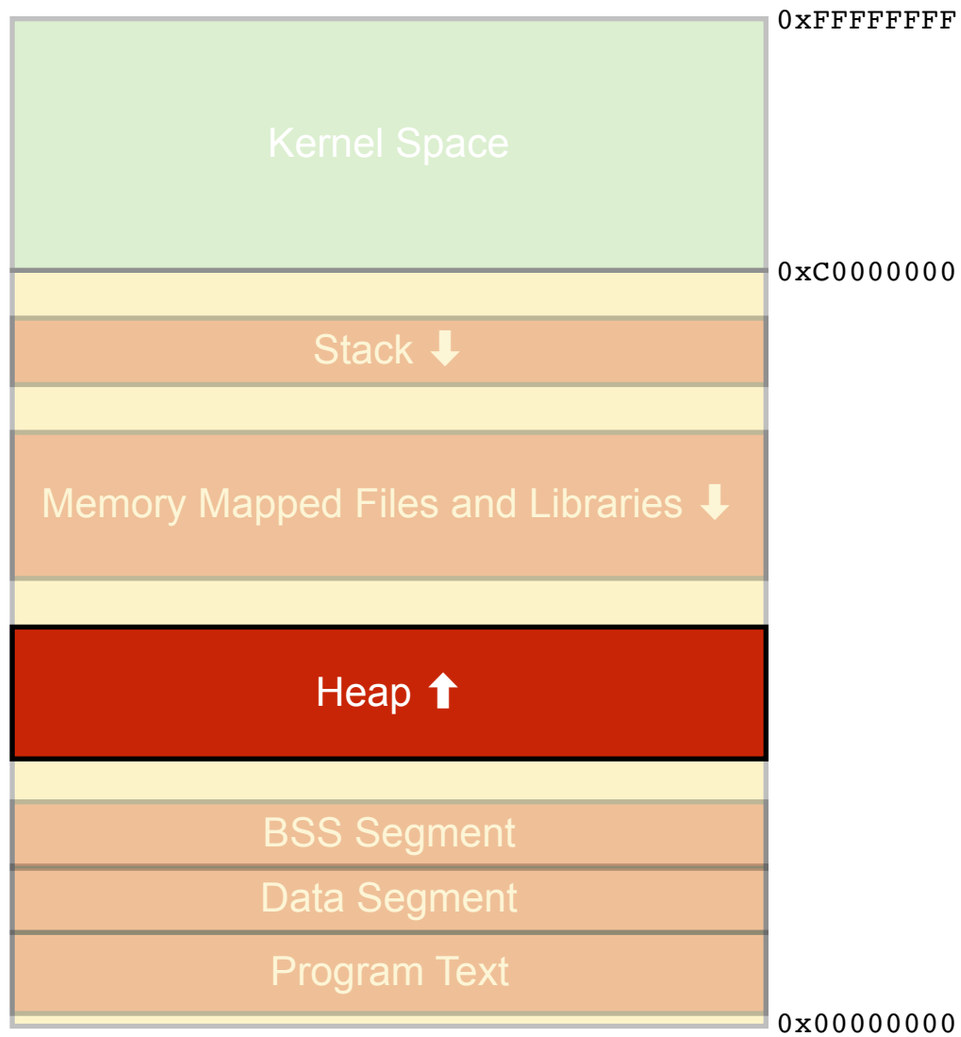


Local variables stored on stack immediately preceding function return address – overflowing local variable space will overwrite return address

CC BY ND

The Heap

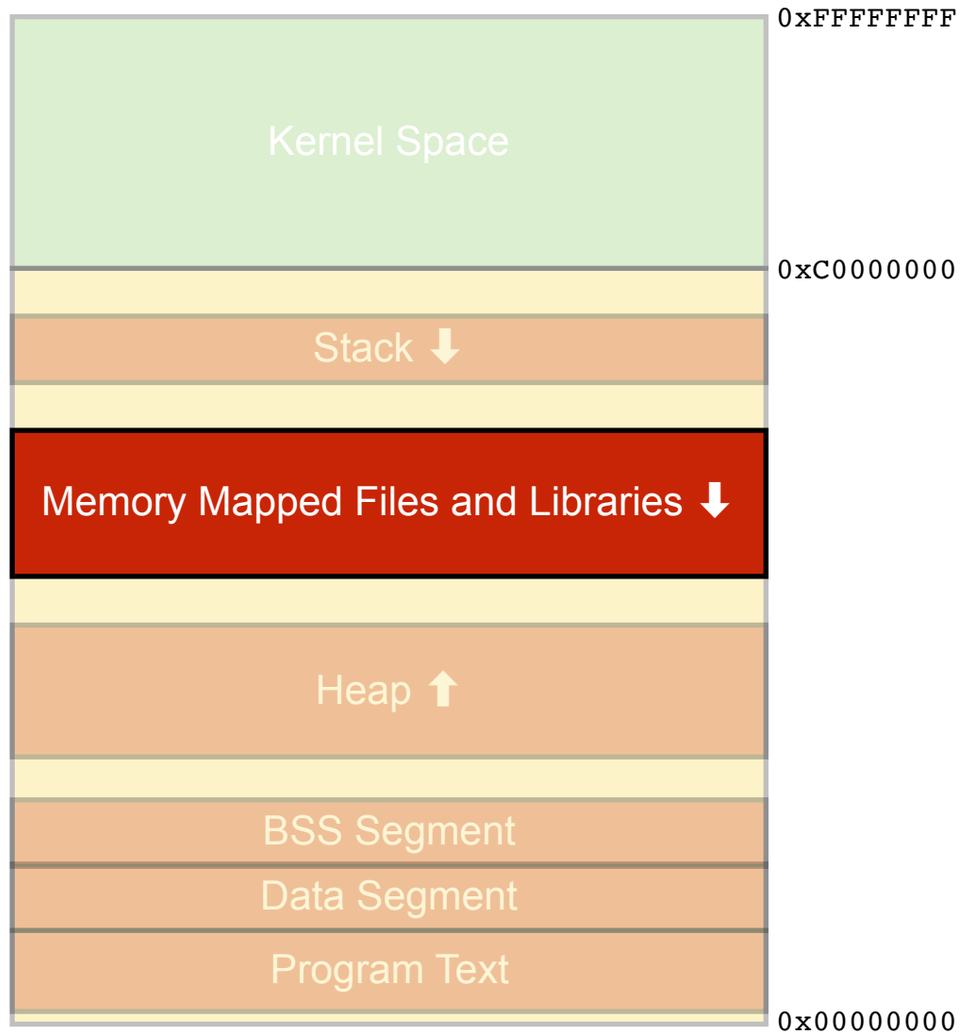
- Explicitly allocated memory located in the heap
  - In C, memory allocated using `malloc()`/`calloc()`
  - In Java, objects allocated using `new`
  - …
- Starts at a low address in memory
- Consecutive allocations placed at increasing addresses in memory
  - Usually consecutive addresses, although 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



Colin Perkins | https://csperkins.org/ | Copyright © 2018

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

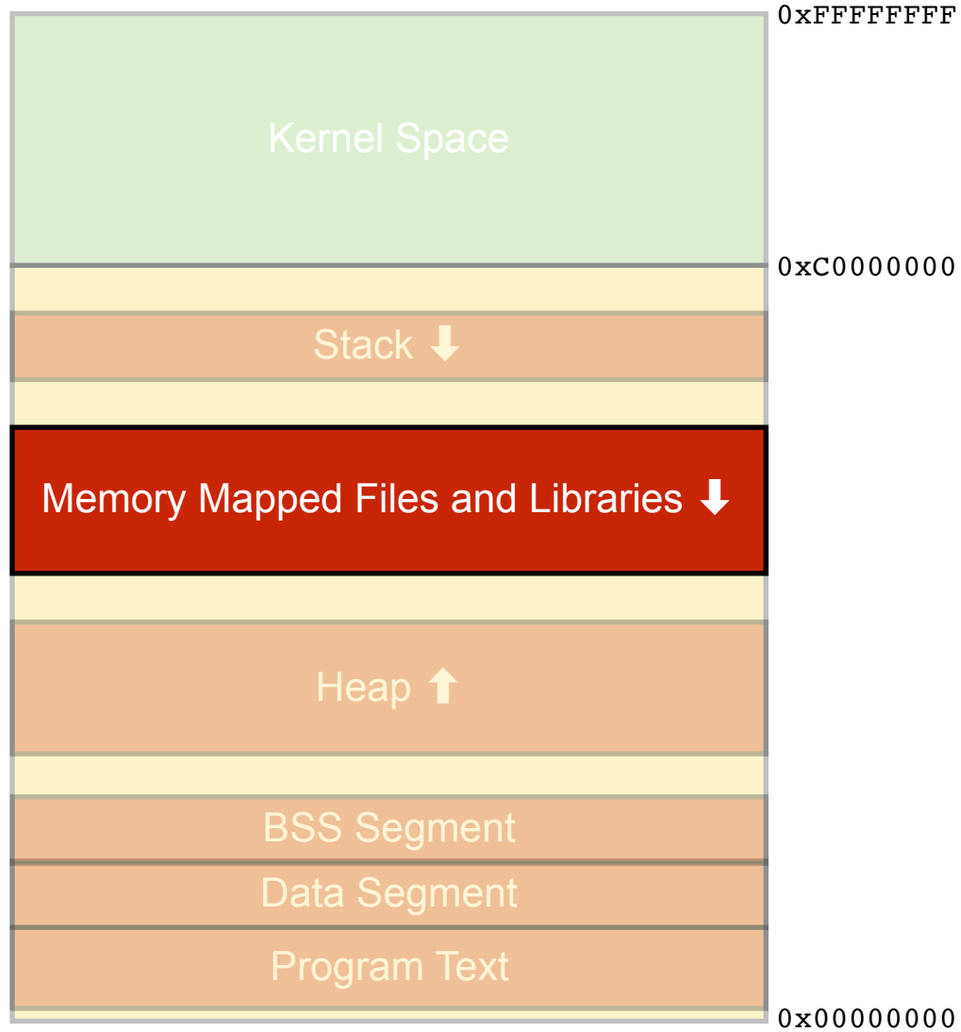


CC BY ND

Shared Libraries

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

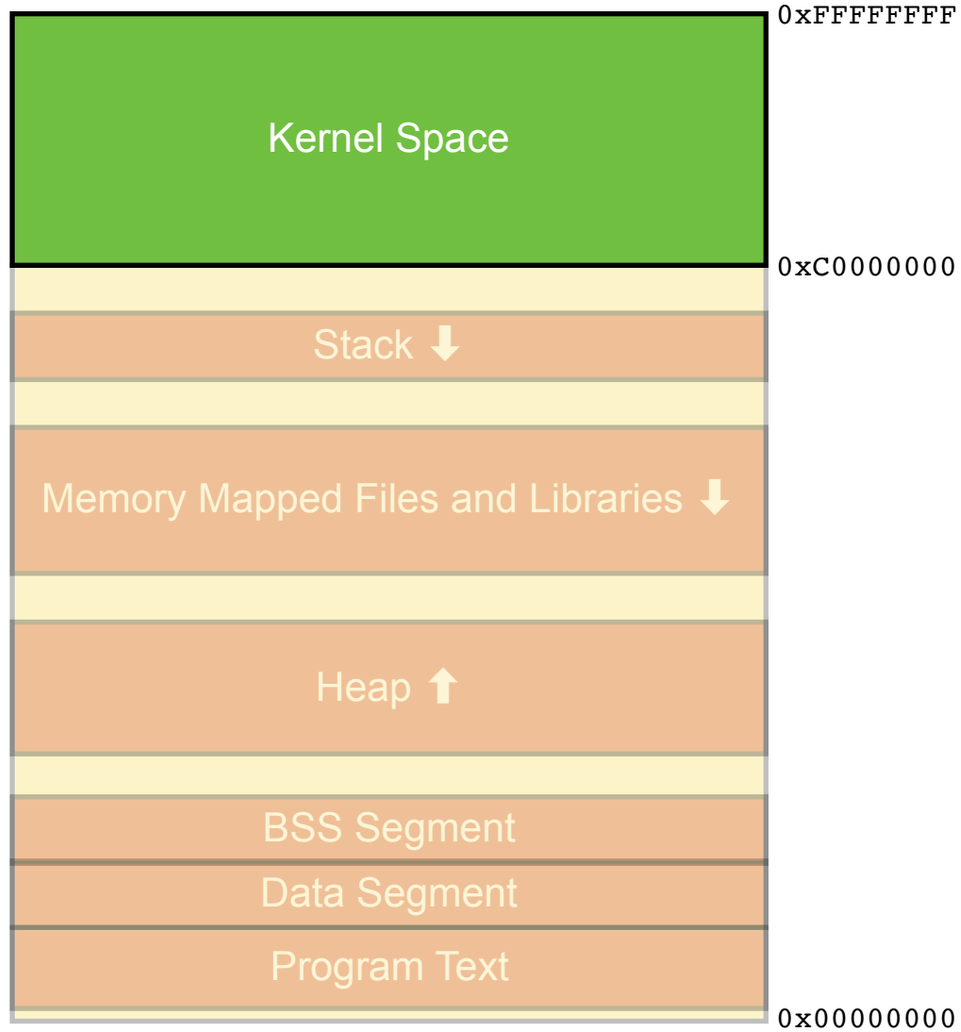
- One copy only in physical memory; virtual address can differ for each process



CC BY ND

The Kernel

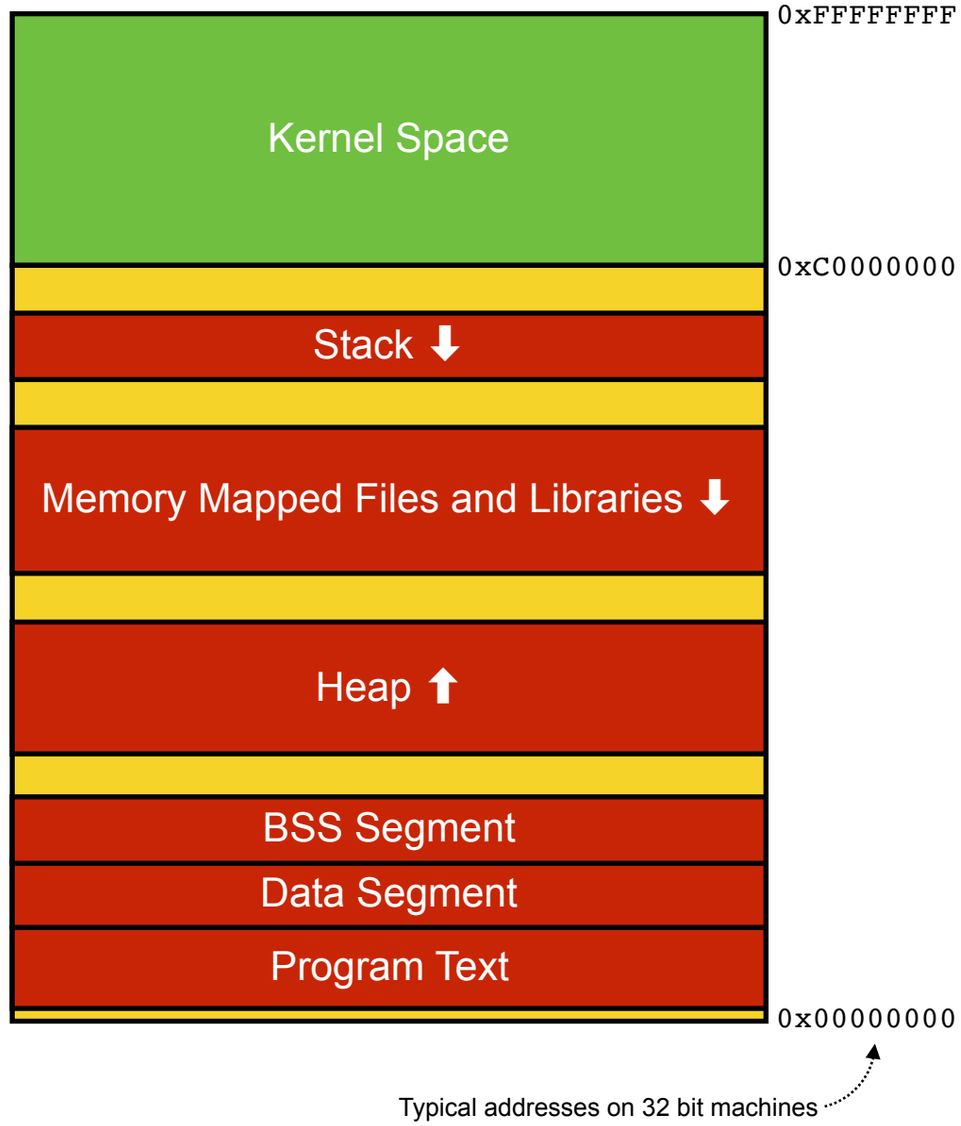
- Operating system kernel owns top part of the address space
  - Not accessible to user-space programs – attempting to read kernel memory gives segmentation violation
  - The `syscall` instruction in x86_64 assembler jumps to the kernel, to one of a set of pre-defined system calls – switches processor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



Colin Perkins | https://csperkins.org/ | Copyright © 2018

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



CC BY ND

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 – after the break, and next lecture

CC BY ND

Region-based Memory Management

Rationale

- Garbage collection (→ lecture 4) tends to have unpredictable timing and high memory overhead
- Manual memory management is too error prone

- Region-based memory management aims for a middle ground between the two approaches
- Safe, predictable timing
- Limited impact on application design

CC BY ND

Stack-based Memory Management

- Automatic allocation/deallocation of variables on the stack is common and efficient:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

static double pi = 3.14159;

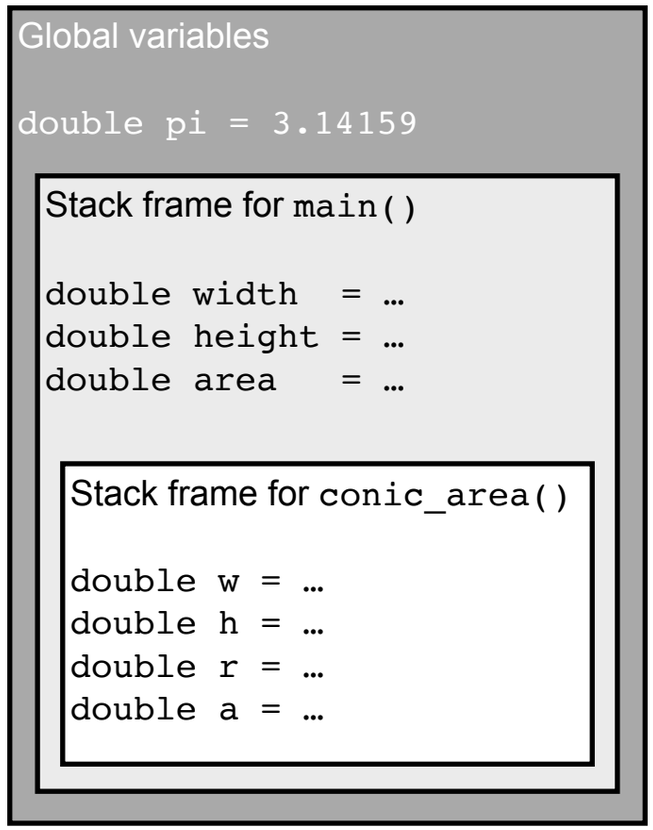
static double conic_area(double w, double h) {
  double r = w / 2.0;
  double a = pi * r * (r + sqrt(h*h + r*r));

  return a;
}

int main() {
  double width  = 3;
  double height = 2;
  double area = conic_area(width, height);

  printf("area of cone = %f\n", area);

  return 0;
}
```



CC BY ND

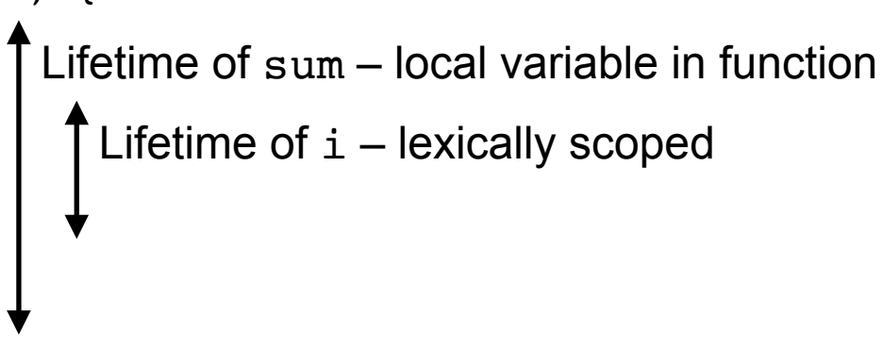
Stack-based Memory Management

- Hierarchy of regions corresponding to call stack:
  - Global variables
  - Local variables in each function
  - Lexically scoped variables within functions

```
double vector_avg(double *vec, int len) {
  double sum = 0;

  for (int i = 0; i < len; i++) {
    sum += vec[i];
  }

  return sum / len;
}
```



- Variables live within regions, and are deallocated at end of region scope

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Stack-based Memory Management

- Limitation: requires data to be allocated on stack
  - Example:

```
int hostname_matches(char *requested, char *host, char *domain) {
  char *tmp = malloc(strlen(host) + strlen(domain) + 2);

  sprintf(tmp, "%s.%s", host, domain);

  if (strcmp(requested, host) == 0) {
    return 1;
  }
  if (strcmp(requested, tmp) == 0) {
    return 1;
  }
  return 0;
}
```

    The local variable `tmp` (pointer of type `char *`) is freed when the function returns; the allocated memory is not freed

- Heap storage has to be managed manually

BY ND

Region-based Memory Allocation

- Stack allocation effective within a narrow domain – can we extend the ideas to manage the heap?
  - Create pointers to heap allocated memory
  - Pointers are stored on the stack and have lifetime matching the stack frame – pointers have type `Box<T>` for a pointer to heap allocated `T`
  - The heap allocation has lifetime matching that of the `Box` – when the `Box` goes out of scope, the heap memory it references is freed
    - i.e., the destructor of the `Box<T>` frees the heap allocated `T`
    - This is RAII, to C++ programmers

- Efficient, but loses generality of heap allocation since ties heap allocation to stack frames

BY ND

Region-based Memory Management

- For effective region-based memory management:
  - Allocate objects with lifetimes corresponding to regions
  - Track object ownership, and *changes of ownership:*
    - What region owns each object at any time
    - Ownership of objects can move between regions
  - Deallocate objects at the end of the lifetime of their owning region
    - Use scoping rules to ensure objects are not referenced after deallocation

- Example: the Rust programming language
  - Builds on previous research with Cyclone language (AT&T/Cornell)
  - Somewhat similar ideas in Microsoft's Singularity operating system

CC BY ND

Returning Ownership of Data

- Returning data from a function causes it to outlive the region in which it was created:

```
const PI: f64 = 3.14159;

fn area_of_cone(w : f64, h : f64) -> f64 {
    let r = w / 2.0;
    let a = PI * r * (r + (h*h + r*r).sqrt());

    return a;
}

fn main() {
    let width  = 3.0;
    let height = 2.0;

    let area = area_of_cone(width, height);

    println!("area = {}", area);
}
```

Lifetime of local variables in `area_of_cone`

Lifetime of `a`

CC BY ND

Returning Ownership of Data

- Runtime must track changes in ownership as data is returned
  - Copies made of stack-allocated local variables; original deallocated, copy has lifetime of stack-allocated local variable in calling function
  - Allows us to return a copy of a `Box<T>` that references a heap allocated value of type `T`
    - Effective with a reference counting implementation
      - Creating the new `Box<T>` temporarily increases the reference count on the heap-allocated `T`
      - The original box is then immediately deallocated, reducing the reference count again
      - (An optimised runtime can eliminate the changes to the reference count)
      - The heap-allocated `T` is deallocated when the box goes out of scope of the outer region
- Allows data to be passed around, if it always has a single owner

CC BY ND

Giving Ownership of Data

```
% cat consume.rs
fn consume(mut x : Vec<u32>) {
  x.push(1);
}

fn main() {
  let mut a = Vec::new();

  a.push(1);
  a.push(2);

  consume(a);

  println!("a.len() = {}", a.len());
}
```

Lifetime of `a`

Ownership of `a` transferred to `consume()`

```
% rustc consume.rs
consume.rs:15:28: 15:29 error: use of moved value: `a` [E0382]
consume.rs:15   println!("a.len() = {}", a.len());
                                         ^
```

- Ownership of parameters passed to a function is transferred to that function
  - Deallocated when function ends, unless it returns the data
  - Data cannot be later used by the calling function – enforced at compile time

CC BY ND

Borrowing Data

```
% cat borrow.rs
fn borrow(mut x : &mut Vec<u32>) {
  x.push(1);
}

fn main() {
  let mut a = Vec::new();

  a.push(1);
  a.push(2);

  borrow(&mut a);

  println!("a.len() = {}", a.len());
}

% rustc borrow.rs
% ./borrow
a.len() = 3
%
```



- Functions can *borrow* references to data owned by an enclosing scope
  - Does not transfer ownership of the data
  - Naïvely safe to use, since live longer than the function
- Can also return references to input parameters passed as references
  - Safe, since these references must live longer than the function

CC BY ND

Problems with Naïve Borrowing

```
% cat borrow.rs
fn borrow(mut x : &mut Vec<u32>) {
  x.push(1);
}

fn main() {
  let mut a = Vec::new();

  a.push(1);
  a.push(2);

  borrow(&mut a);

  println!("a.len() = {}", a.len());
}

% rustc borrow.rs
% ./borrow
a.len() = 3
%
```

- The `borrow()` function changes the contents of the vector
- But – it cannot know whether it is safe to do so
  - In this example, it is safe
  - If `main()` was iterating over the contents of the vector, changing the contents might lead to elements being skipped or duplicated, or to a result to be calculated with inconsistent data
- Known as *iterator invalidation*

BY ND

Safe Borrowing

- Rust has two kinds of pointer:
  - `&T` – a shared reference to an immutable object of type `T`
  - `&mut T` – a unique reference to a mutable object of type `T`
- Runtime system controls pointer ownership and use
  - An object of type `T` can be referenced by one or more references of type `&T`, or by exactly one reference of type `&mut T`, but not both
  - Cannot get an `&mut T` reference to data of type `T` that is marked as immutable (i.e., via an `&T` reference)
- Allows functions to safely borrow objects – without needing to give away ownership

- To change an object:
  - You either own the object, and it is not marked as immutable; or
  - You have the only `&mut` reference to it
- Prevents iterator invalidation
  - The iterator requires an `&T` reference, so other code can't get a mutable reference to the contents to change them:

```
fn main() {
  let mut data = vec![1, 2, 3, 4, 5, 6];
  for x in &data {
    data.push(2 * x);
  }
}
```

fails, since push takes an `&mut` reference

- enforced at compile time

CC BY ND

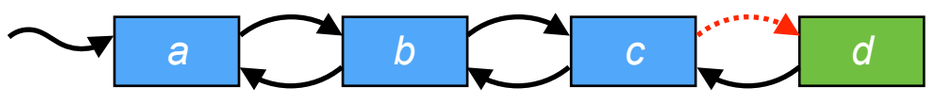
Benefits

- Type system tracks ownership, turning run-time bugs into compile-time errors:
  - Prevents memory leaks and use-after-free bugs
  - Prevents iterator invalidation
  - Prevents race conditions with multiple threads – borrowing rules prevent two threads from getting references to a mutable object
- Efficient run-time behaviour – timing and memory usage are predictable

CC BY ND

Limitations of Region-based Systems

- Can’t express cyclic data structures
  - E.g., can’t build a doubly linked list:



Can’t get mutable reference to *c* to add the link to *d*, since already referenced by *b*

  - Many languages offer an escape hatch from the ownership rules to allow these data structures (e.g., raw pointers and `unsafe` in Rust)

- Can’t express shared ownership of mutable data
  - Usually a good thing, since avoids race conditions
  - Rust has `RefCell<T>` that dynamically enforces the borrowing rules (i.e., allows upgrading a shared reference to an immutable object into a unique reference to a mutable object, if it was the only such shared reference)
  - Raises a run-time exception if there could be a race condition, rather than preventing it at compile time

CC BY ND

Limitations of Region-based Systems

- Forces consideration of object ownership early and explicitly
  - Generally good practice, but increases conceptual load early in design process – may hinder exploratory programming

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Summary

- Region-based memory management with strong ownership and borrowing rules
  - Efficient and predictable behaviour
  - Strong correctness guarantees prevent many common bugs
  - Constrains the type of programs that can be written
- Further reading:
  - D. Grossman et al., “Region-based memory management in Cyclone”, Proc. ACM PLDI, Berlin, Germany, June 2002. DOI:10.1145/512529.512563
  - You are not expected to read/understand section 4
  - What was Cyclone? Did the project’s goals make sense?
  - How does the region-based memory management system described differ from that outlined in the lecture?
  - Interactions with the garbage collector?
  - Other features added to C?
  - Ease of porting C code? Performance?
  - Does it make sense to try to extend C with region-based memory management?



Region-Based Memory Management in Cyclone *

Dan Grossman Greg Morrisett Trevor Jim†
Michael Hicks Yanling Wang James Cheney

Computer Science Department
Cornell University
Ithaca, NY 14853
{danieljg,jgm,mhicks,wangyl,jcheney}@cs.cornell.edu

†AT&T Labs Research
180 Park Avenue
Florham Park, NJ 07932
trevor@research.att.com

ABSTRACT

Cyclone is a type-safe programming language derived from C. The primary design goal of Cyclone is to let programmers control data representation and memory management without sacrificing type-safety. In this paper, we focus on the region-based memory management of Cyclone and its static typing discipline. The design incorporates several advancements, including support for region subtyping and a coherent integration with stack allocation and a garbage collector. To support separate compilation, Cyclone requires programmers to write some explicit region annotations, but a combination of default annotations, local type inference, and a novel treatment of region effects reduces this burden. As a result, we integrate C idioms in a region-based framework. In our experience, porting legacy C to Cyclone has required altering about 8% of the code; of the changes, only 6% (of the 8%) were region annotations.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*

General Terms

Languages

1. INTRODUCTION

Many software systems, including operating systems, device drivers, file servers, and databases require fine-grained control over data representation (e.g., field layout) and resource management (e.g., memory management). The *de facto* language for coding such systems is C. However, in providing low-level control, C admits a wide class of dangerous — and extremely common — safety violations, such as incorrect type casts, buffer overruns, dangling-pointer dereferences, and space leaks. As a result, building large systems in C, especially ones including third-party extensions, is perilous. Higher-level, type-safe languages avoid these drawbacks, but in so doing, they often fail to give programmers the control needed in low-level systems. Moreover, porting or extending legacy code is often prohibitively expensive. Therefore, a safe language at the C level of abstraction, with an easy porting path, would be an attractive option.

Toward this end, we have developed *Cyclone* [6, 19], a language designed to be very close to C, but also safe. We have written or ported over 110,000 lines of Cyclone code, including the Cyclone compiler, an extensive library, lexer and parser generators, compression utilities, device drivers, a multimedia distribution overlay network, a web server, and many smaller benchmarks. In the process, we identified many common C idioms that are usually safe, but which the C type system is too weak to verify. We then augmented the language with modern features and types so that programmers can still use the idioms, but have safety guarantees.

For example, to reduce the need for type casts, Cyclone has features like parametric polymorphism, subtyping, and tagged unions. To prevent bounds violations without making hidden data-representation changes, Cyclone has a variety of pointer types with different compile-time invariants and associated run-time checks. Other projects aimed at making legacy C code safe have addressed these issues with somewhat different approaches, as discussed in Section 7.

In this paper, we focus on the most novel aspect of Cyclone: its system for preventing dangling-pointer dereferences and space leaks. The design addresses several seemingly conflicting goals. Specifically, the system is:

- *Sound:* Programs never dereference dangling pointers.
- *Static:* Dereferencing a dangling pointer is a compile-time error. No run-time checks are needed to determine if memory has been deallocated.
- *Convenient:* We minimize the need for explicit programmer annotations while supporting many C idioms. In particular, many uses of the addresses of local variables require no modification.

*This research was supported in part by Sloan grant BR-3734; NSF grant 9875536; AFOSR grants F49620-00-1-0198, F49620-01-1-0298, F49620-00-1-0209, and F49620-01-1-0312; ONR grant N00014-01-1-0968; and NSF Graduate Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'02, June 17-19, 2002, Berlin, Germany.
Copyright 2002 ACM 1-58113-463-0/02/0006 ...$5.00.

Colin Perkins | https://csperkins.org/ | Copyright © 2018