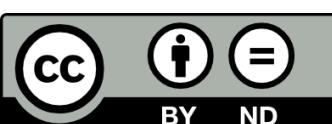


Advanced Topics in Systems Programming

Advanced Systems Programming (H/M)

Lecture 1



Course Team and Materials

- Course team:
 - Dr Colin Perkins (lectures)
 - Email: colin.perkins@glasgow.ac.uk
 - Office: S101b, Lilybank Gardens
 - Dr Anna Lito Michala (labs)
 - Email: AnnaLito.Michala@glasgow.ac.uk
 - Office: 405, Sir Alwyn Williams Building
- Course materials are available on Moodle



Rationale

- Technology shift: desktop PC → laptop, tablet, smartphone, cloud; mobile, power-aware, concurrent, real-time, connected

Rationale

- Technology shift: desktop PC → laptop, tablet, smartphone, cloud; mobile, power-aware, concurrent, real-time, connected
- But still programmed in C, running some variant of Unix – technology stack that's becoming increasing limiting

Rationale

- Technology shift: desktop PC → laptop, tablet, smartphone, cloud; mobile, power-aware, concurrent, real-time, connected
- But still programmed in C, running some variant of Unix – technology stack that's becoming increasingly limiting
- This course will explore new techniques for safer, more effective, systems programming
 - Programming in an unmanaged environment, where data layout and performance matter
 - Operating systems kernels, device drivers, low-level networking code

Aims and Objectives

- The course aims to explore the features of modern programming languages and operating systems that can ease the challenges of systems programming, considering type systems and run-time support.
- It will review the research literature on systems programming and operating system interfaces, discuss the limitations of deployed systems, and consider how systems programming might evolve to address the challenges of supporting modern computing systems.
- Particular emphasis will be placed on system correctness and secure programming, to ensure the resulting systems are safe to use in an adversarial environment.

Intended Learning Outcomes (1/2)

- By the end of the course, students should be able:
 - To discuss the advantages and disadvantages of C as a systems programming language, and to compare and contrast this with a modern systems programming language, for example Rust; to discuss the role of the type system, static analysis, and verification tools in systems programming, and show awareness of how to model system properties using the type system to avoid errors;

Intended Learning Outcomes (1/2)

- By the end of the course, students should be able:
 - To discuss the advantages and disadvantages of C as a systems programming language, and to compare and contrast this with a modern systems programming language, for example Rust; to discuss the role of the type system, static analysis, and verification tools in systems programming, and show awareness of how to model system properties using the type system to avoid errors;
 - To discuss the challenges of secure low-level programming and write secure code in a modern systems programming language to perform systems programming tasks such as parsing hostile network input; show awareness of security problems in programs written in C;

Intended Learning Outcomes (1/2)

- By the end of the course, students should be able:
 - To discuss the advantages and disadvantages of C as a systems programming language, and to compare and contrast this with a modern systems programming language, for example Rust; to discuss the role of the type system, static analysis, and verification tools in systems programming, and show awareness of how to model system properties using the type system to avoid errors;
 - To discuss the challenges of secure low-level programming and write secure code in a modern systems programming language to perform systems programming tasks such as parsing hostile network input; show awareness of security problems in programs written in C;
 - To discuss the advantages and disadvantages of integrating automatic memory management with the operating system/runtime, to understand the operation of popular garbage collection algorithms and alternative techniques for memory management, and know when it might be appropriate to apply such techniques and managed run-times to real-time systems and/or operating systems;

Intended Learning Outcomes (2/2)

- By the end of the course, students should be able:
 - To understand the impact of heterogeneous multicore systems on operating systems, compare and evaluate different programming models for concurrent systems, their implementation, and their impact on operating systems; and

Intended Learning Outcomes (2/2)

- By the end of the course, students should be able:
 - To understand the impact of heterogeneous multicore systems on operating systems, compare and evaluate different programming models for concurrent systems, their implementation, and their impact on operating systems; and
 - To construct and/or analyse simple programs to demonstrate understanding of novel techniques for memory management and/or concurrent programming, to understand the trade-offs and implementation decisions.

Pre-requisites

- You are expected to be familiar with the C programming language, and to understand the basics of operating systems and networking
- A conceptual understanding of functional programming is assumed, but Haskell programming is not required
- This broadly corresponds to the material in the following courses:
 - Systems Programming (H)
 - Operating Systems (H)
 - Networked Systems (H)
 - Functional Programming (H)

Course Structure

Week	10:00-12:00 – Group Discussion	13:00-14:00 – Lab Sessions
1	#1: Introduction	#1: Systems Programming Languages
2	#2: Systems Programming	#2: Challenges in Systems Programming: Energy Efficiency
3	#3: Types and Systems Programming	#3: Introducing the Rust Programming Language
4	#4: Type-based Modelling and Design	#4: Types and Traits
5	#5: Resource Ownership and Memory Management	#5: State Machines
6	#6: Garbage Collection	#6: Ownership, Pointers, and Memory
7	#7: Concurrency	#7: Closures and Concurrency
8	#8: Coroutines and Asynchronous Programming	#8: Coroutines and Asynchronous Code
9	#9: Security Considerations	
10	#10: Future Directions	

Course Structure: Lectures

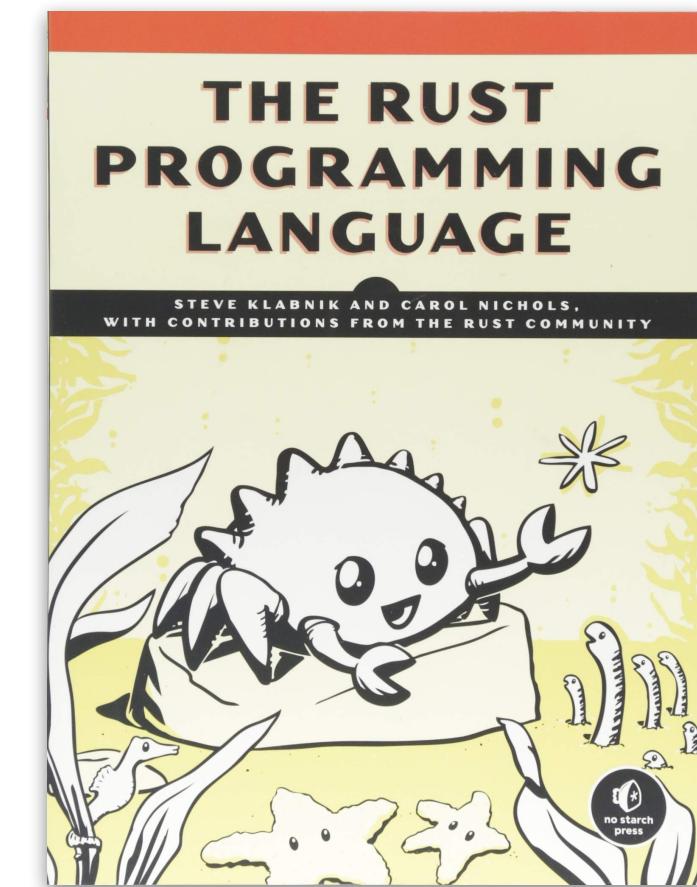
Week	10:00-12:00 – Group Discussion	13:00-14:00 – Lab Sessions
1	#1: Introduction	#1: Systems Programming Languages
2	#2: Systems Programming	#2: Challenges in Systems Programming: Energy Efficiency
3	#3: Types and Systems Programming	#3: Introducing the Rust Programming Language
4	#4: Type-based Modelling and Design	#4: Types and Traits <ul style="list-style-type: none">Lecture recordings will be made available ahead of time, and each is accompanied by discussion questionsTimetabled session on Monday mornings is for discussion: watch the lecture and think about the questions before the timetabled slot
5	#5: Resource Ownership and Memory Management	
6	#6: Garbage Collection	
7	#7: Concurrency	
8	#8: Coroutines and Asynchronous Programming	#8: Coroutines and Asynchronous Code
9	#9: Security Considerations	
10	#10: Future Directions	

Course Structure: Labs

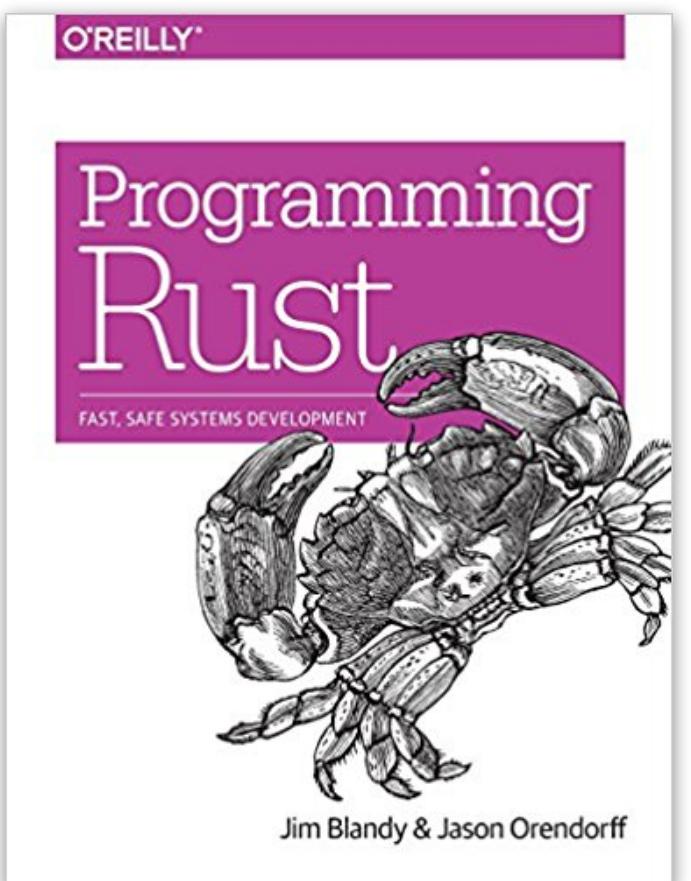
Week	10:00-12:00 – Group Discussion	13:00-14:00 – Lab Sessions
1	#1: Introduction	#1: Systems Programming Languages
2	#2: Systems Programming	#2: Challenges in Systems Programming: Energy Efficiency
<ul style="list-style-type: none">Timetabled sessions on Monday afternoons are for discussion and support with the lab exercises – try to solve the exercise, and think of questions you might need to ask, before the timetabled support slotLab exercises are tested on the student Linux servers (stlinux01 to stlinux08.dcs.gla.ac.uk) provided by the School but should run on any system with a recent version of the Rust programming language installed		#3: Introducing the Rust Programming Language
8	#8: Coroutines and Asynchronous Programming	#4: Types and Traits
9	#9: Security Considerations	#5: State Machines
10	#10: Future Directions	#6: Ownership, Pointers, and Memory
		#7: Closures and Concurrency
		#8: Coroutines and Asynchronous Code

Recommended Reading (1/2)

- The Rust Programming language (<https://rust-lang.org/>) will be used to illustrate principles of ownership, memory management, and type-driven programming – **read one of these books**
- You are expected to learn the basics of programming in Rust



Steve Klabnik and Carol Nichols, "The Rust Programming Language", 2nd Edition, 2018, ISBN 978-1-59327-828-1 ([Amazon](#), [free online edition](#)).



Jim Blandy and Jason Orendorff, "Programming Rust", O'Reilly, 2018, ISBN 978-1-491-92728-1 ([Amazon](#)).
Free access via University Library: <https://tinyurl.com/y7gxx8dc>

Recommended Reading (2/2)

- Research papers and blog posts will be cited to illustrate some concepts
 - Citations with URLs and/or DOIs (<http://dx.doi.org/>) provided
 - All papers are accessible at no cost from the campus network or VPN – **do not pay to access research papers**
- **You are expected to read this material**
 - Critical reading of a research paper requires practice; read in a structured manner, not end-to-end, thinking about the material as you go; focus on the concepts rather than details
 - See <http://www.eecs.harvard.edu/~michaelm/postscripts/ReadPaper.pdf>
 - Realise that research papers are written to explore new ideas
 - Some will be good ideas, some less so; some will be interesting but infeasible; what's impractical today might be important tomorrow – changes in technology and/or society can change what's feasible/desirable
 - Think and judge for yourself!

Assessment

- This is a level H course, worth 10 credits
- Assessment is by examination (80%) and coursework (20%)
 - Sample exam paper, with worked answers, is available
 - Material from the lectures, labs, **and cited papers and blog posts** is examinable
 - Aim is to test your understanding of the material, not to test your memory of all the details
 - Explain why – don't just recite what
- Assessed coursework:

Coursework	Date Set:	Date Due:	Weighting:	Topic:
Exercise 1	Lecture 5	Lecture 7	10%	Memory Management (essay)
Exercise 2	Lecture 7	Lecture 9	10%	Concurrent Programming in Rust (code)

Advanced Topics in Systems Programming

- Lecture material starts in week 2 of the semester – **watch lecture 2 recording and think about the discussion questions** before then
- Labs start in week 1 – **review the lab 1 handout and start on the exercise**



Systems Programming

Advanced Systems Programming (H/M)
Lecture 2



Colin Perkins | <https://csperrkins.org/> | Copyright © 2020 University of Glasgow | 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.

Systems Programming

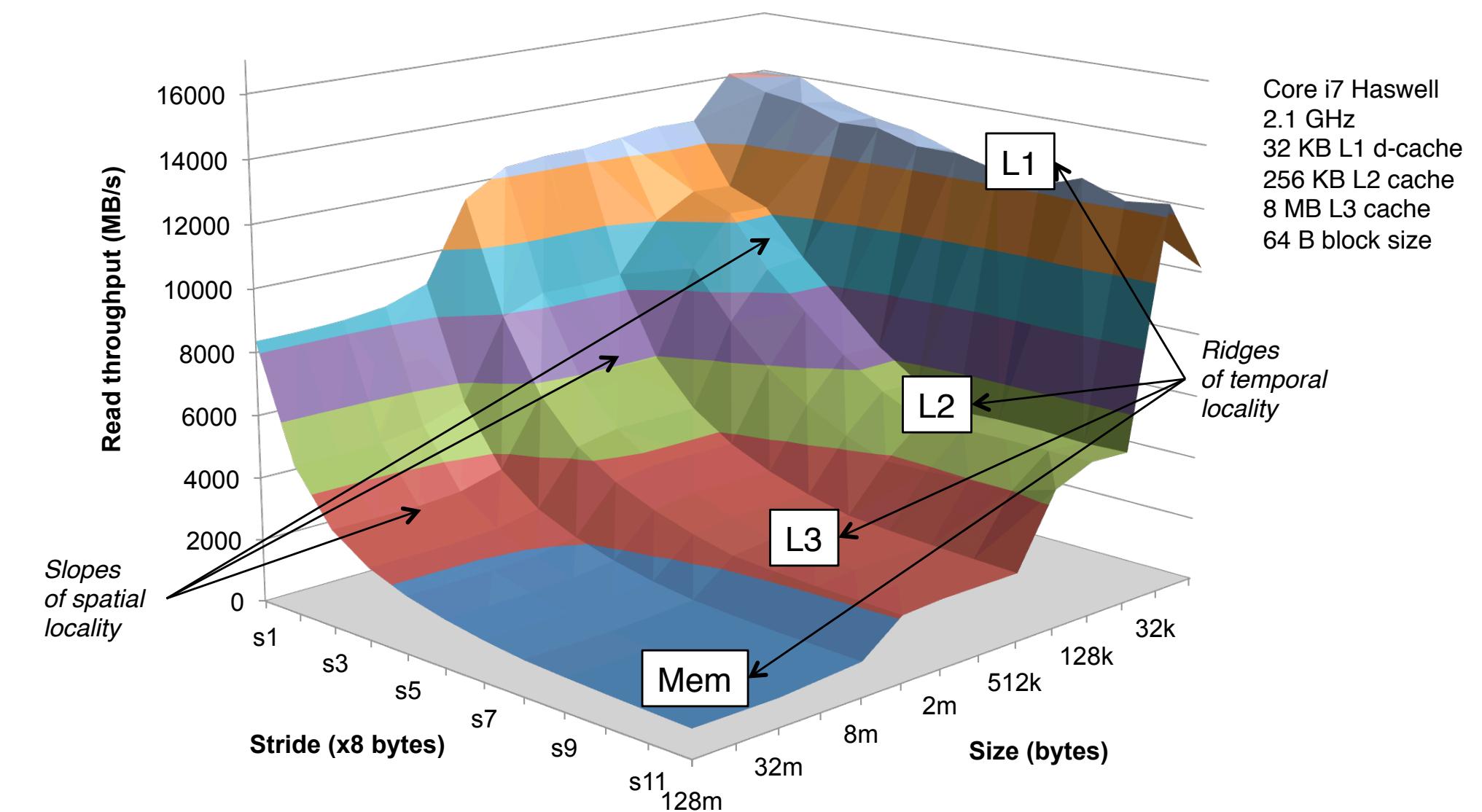
- What is systems programming?
- The state of the art
- Challenges and limitations
- Next steps in systems programming

What is Systems Programming?

- Systems programs comprise infrastructure components: operating systems, device drivers, network protocols, and services
- They tend to be constrained by:
 - Memory management and data representation
 - I/O operations
 - Management of shared state

Memory Management and Data Representation

- Predictability:
 - Timing must be bounded for real-time applications
 - Bounds on memory usage
- Data locality:
 - Cache line sharing impacts performance
 - Ensuring data is aligned and packed into cache lines for high performance
- Data representation:
 - Device drivers with fixed-layout control registers
 - Network protocol implementations must conform to specified packet layout
- Systems programming languages offer control of memory management and data representation – others languages lack such controls

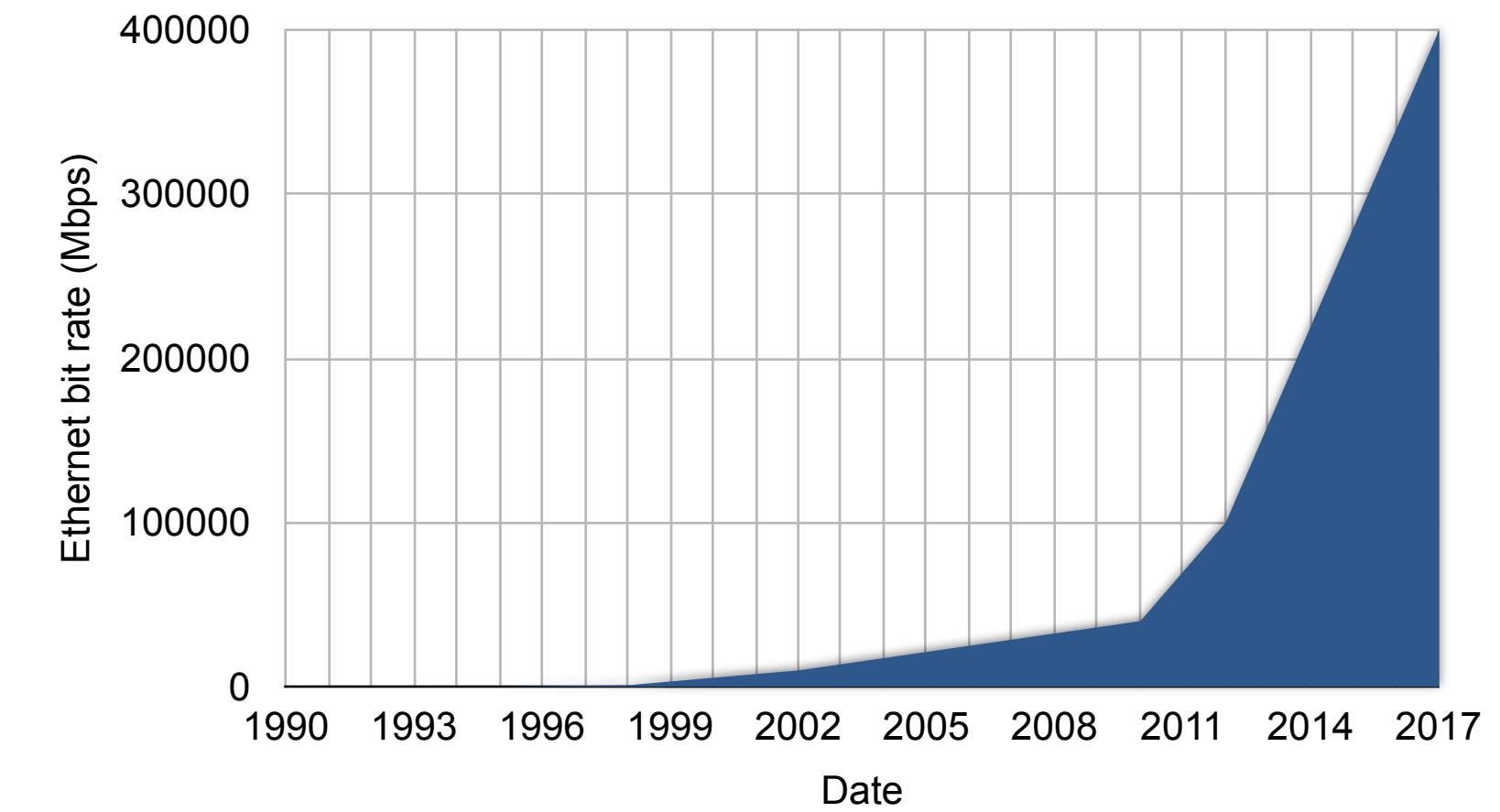


Smaller values of **stride** indicate data with better spatial locality; **size** is the total amount of data accessed

Source: Bryant & O'Hallaron, "Computer Systems: A Programmer's Perspective", 3rd Edition, Pearson, 2016, Fig. 6.41. <http://csapp.cs.cmu.edu/3e/figures.html> (Permission granted for lecture use with attribution)

I/O Operations

- Network performance increasingly a bottleneck:
 - Chart shows Ethernet bit rate over time – wireless links follow a similar curve
 - Closely tracking exponential growth over time – unlike CPU speed, where growth mostly stopped mid-2000s
 - MTU remains constant but packet rate increases; fewer cycles to process each packet
- SSD performance on a similar trend for file system access
- I/O performance of systems software critical to overall system performance
 - Device drivers, network protocol stack, file system

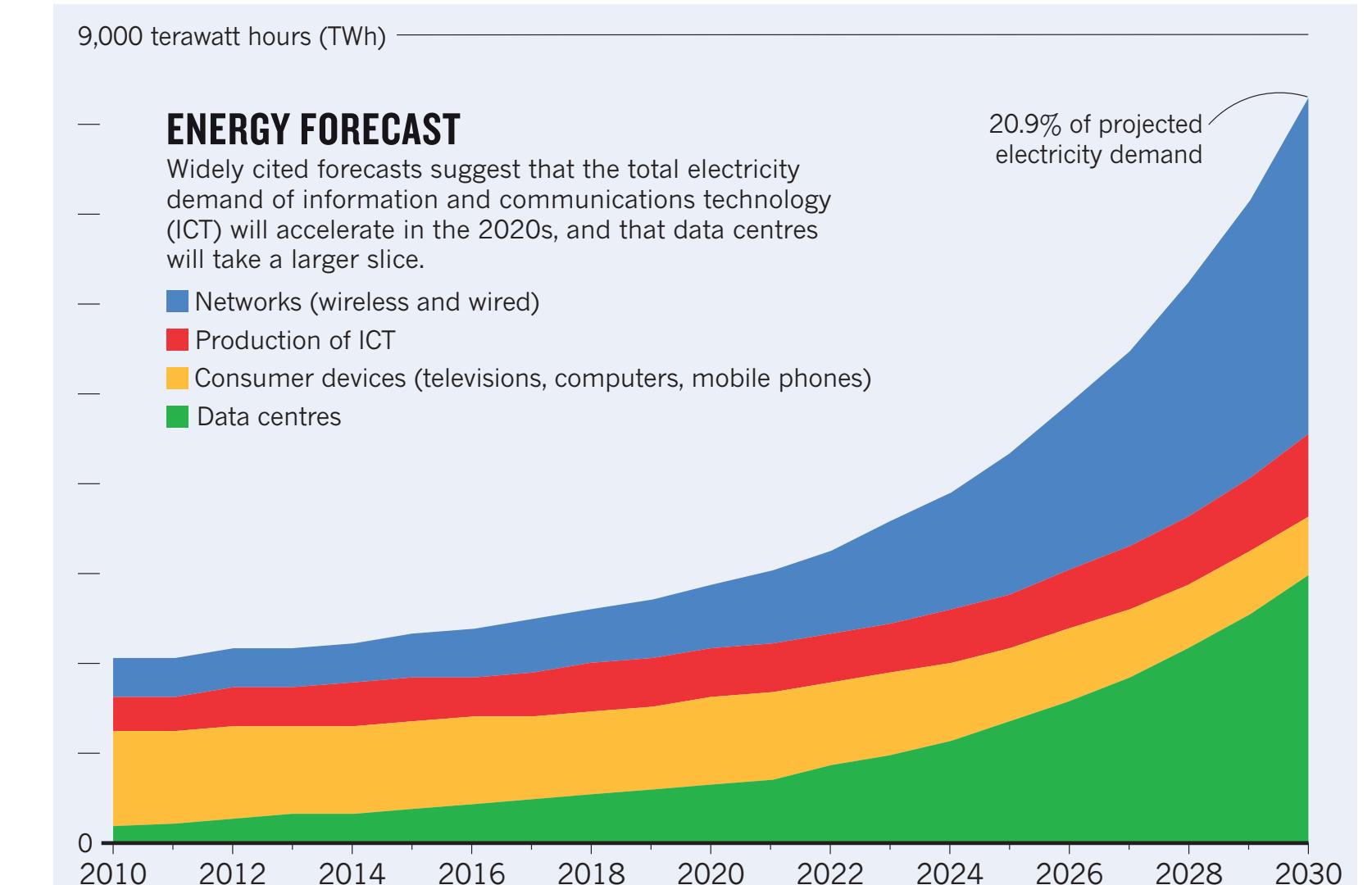


Management of Shared State

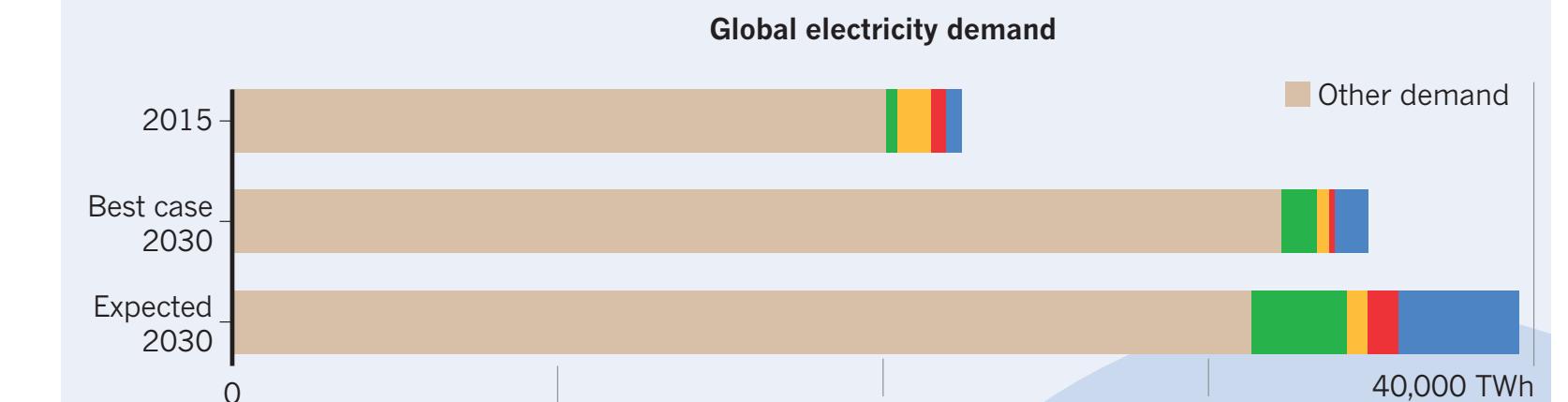
- Systems programs responsible for coordinating shared mutable state:
 - State shared across layers/between kernel and applications
 - Data structures for zero-copy networking
 - Header processing and state for the TCP stack
 - State shared between threads
 - Internal state of the kernel
 - File systems
 - Network code
 - Highly performance critical
- Systems programming languages allow sharing data between layers and/or threads – other languages disallow/discourage such sharing

Performance

- Systems infrastructure performance fundamentally affects overall system and application performance
 - Mobile devices have limited battery life
 - Data centre efficiency and power consumption
- Systems components often the bottleneck in terms of overall performance and power efficiency
 - Simply because they're the basis on which the higher-level components depend



The chart above is an 'expected case' projection from Anders Andrae, a specialist in sustainable ICT. In his 'best case' scenario, ICT grows to only 8% of total electricity demand by 2030, rather than to 21%.



INTERNET EXPLOSION

Internet traffic* is growing exponentially, and reached more than a zettabyte (ZB, 10^{21} bytes) in 2017.



*Traffic to and from data centres. [†]TB, terabyte (10^{12} bytes); PB, petabyte (10^{15} bytes); EB, exabyte (10^{18} bytes).

Source: N. Jones, "The Information Factories", Nature, v.561, p.163–166, Sep. 2018.DOI:[10.1038/d41586-018-06610-y](https://doi.org/10.1038/d41586-018-06610-y)

Systems Programming

- Systems programming languages offer low-level control of data representation, memory management, I/O, and sharing
- They are high-performance – concrete rather than high abstraction

Programming Language Challenges in Systems Codes

Why Systems Programmers Still Use C, and What to Do About It

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Department of Computer Science
Johns Hopkins University
shap@cs.jhu.edu

Abstract

There have been major advances in programming languages over the last 20 years. Given this, it seems appropriate to ask why systems programmers continue to largely ignore these languages. What are the deficiencies in the eyes of the systems programmers? How have the efforts of the programming language community been misdirected (from their perspective)? What can/should the PL community do about this?

As someone whose research straddles these areas, I was asked to give a talk at this year's PLOS workshop. What follows are my thoughts on this subject, which may or not represent those of other systems programmers.

1. Introduction

Modern programming languages such as ML [16] or Haskell [17] provide newer, stronger, and more expressive type systems than systems programming languages such as C [15, 13] or Ada [12]. Why have they been of so little interest to systems developers, and what can/should we do about it?

As the primary author of the EROS system [18] and its successor Coyotos [20], both of which are high-performance microkernels, it seems fair to characterize myself primarily as a hardcore systems programmer and security architect. However, there are skeletons in my closet. In the mid-1980s, my group at Bell Labs developed one of the first large commercial C++ applications — perhaps the first. My early involvement with C++ includes the first book on reusable C++ programming [21], which is either not well known or has been graciously disregarded by my colleagues.

In this audience I am tempted to plead for mercy on the grounds of youth and ignorance, but having been an active

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.

PLOS 2006, Oct. 22, 2006, San Jose, California, United States
Copyright © 2006 ACM 1-59593-577-0/10/2006... \$5.00

advocate of C++ for so long this entails a certain degree of *chutzpah*.¹ There is hope. Microkernel developers seem to have abandoned C++ in favor of C. The book is out of print in most countries, and no longer encourages deviant coding practices among susceptible young programmers.

A Word About BitC Brewer *et al.*'s cry that *Thirty Years is Long Enough*, [6] resonates. It really is a bit disturbing that we are still trying to use a high-level assembly language created in the early 1970s for critical production code 35 years later. But Brewer's lament begs the question: why has no viable replacement for C emerged from the programming languages community? In trying to answer this, my group at Johns Hopkins has started work on a new programming language: BitC. In talking about this work, we have encountered a curious blindness from the PL community.

We are often asked "Why are you building BitC?" The tacit assumption seems to be that if there is nothing fundamentally new in the language it isn't interesting. The BitC goal isn't to invent a new language or any new language concepts. It is to integrate existing concepts with advances in prover technology, and reify them in a language that allows us to build stateful low-level systems codes that we can reason about in varying measure using automated tools. The feeling seems to be that everything we are doing is straightforward (read: uninteresting). Would that it were so.

Systems programming — and BitC — are fundamentally about engineering rather than programming languages. In the 1980s, when compiler writers still struggled with inadequate machine resources, engineering considerations were respected criteria for language and compiler design, and a sense of "transparency" was still regarded as important.² By the time I left the PL community in 1990, respect for engineering and pragmatics was fast fading, and today it is all but gone. The concrete syntax of Standard ML [16] and Haskell [17] are every bit as bad as C++. It is a curious measure of the programming language community that nobody cares. In our pursuit of type theory and semantics,

¹ *Chutzpah* is best defined by example. Chutzpah is when a person murders both of their parents and then asks the court for mercy on the grounds that they are an orphan.

² By "transparent," I mean implementations in which the programmer has a relatively direct understanding of machine-level behavior.

J. Shapiro, "Programming language challenges in systems codes: why systems programmers still use C, and what to do about it", Workshop on Programming Languages and Operating Systems, San Jose, CA, October 2006. DOI:10.1145/1215995.1216004



Systems Programming

- **What is systems programming?**
- The state of the art
- Challenges and limitations
- Next steps in systems programming

The State of the Art

- What is the state of the art in operating systems and systems programming?

The State of the Art

- Most devices run some variant of Unix as their operating system, and are programmed in C
 - Original version of Unix written in assembly for PDP-7 minicomputer in 1969
 - Ported to the PDP-11/40 in early 1970s, re-writing into C at that language was developed
 - “The PDP-11/40 was designed to fit a broad range of applications, from small stand alone situations where the computer consists of only 8K of memory and a processor, to large multi-user, multi-task applications requiring up to 124K of addressable memory space. Among its major features are a fast central processor with a choice of floating point and sophisticated memory management, both of which are hardware options.”
<https://pdos.csail.mit.edu/6.828/2005/readings/pdp11-40.pdf>

- macOS, iOS, Linux and Android are moderns variants and reimplementations of Unix
- This has proven surprisingly resilient and portable – but is it still the right model?



<https://dave.cheney.net/2017/12/04/what-have-we-learned-from-the-pdp-11>
Image credit: Dennis Ritchie

Unix and C: Strengths

- Unix gained popularity due to portability and ease of source code access, but also:
 - Small, relatively consistent set of API calls
 - Low-level control
 - Robust and high performance
 - Easy to understand and extend
- Portability was due to the C programming language
 - Simple, easy to understand, easy to port to new architectures
 - Explicit pointers, memory allocation, control of data representation
 - Uniform treatment of memory, device registers, and data structures – easy to write device drivers, network protocols, and interface with external formats
 - Weak type system allows aliasing and sharing

```
struct {  
    short errors      : 4;  
    short busy       : 1;  
    short unit_sel   : 3  
    short done       : 1;  
    short irq_enable : 1  
    short reserved   : 3  
    short dev_func   : 2;  
    short dev_enable : 1;  
} ctrl_reg;  
  
int enable_irq(void)  
{  
    ctrl_reg *r = 0x80000024;  
    ctrl_reg tmp;  
  
    tmp = *r;  
    if (tmp.busy == 0) {  
        tmp.irq_enable = 1;  
        *r = tmp;  
        return 1;  
    }  
    return 0;  
}
```

Example: hardware access in C

Unix and C: Weaknesses

- Unix APIs reflect 1970s/1980s minicomputer architectures
 - Sockets and file system APIs significant performance bottlenecks
 - Security architecture insufficiently flexible
 - No portable APIs for mobility, power management, etc.
 - Assumes professional, interactive, systems administration
- C programming language
 - Limited concurrency support → memory model for pthreads poorly supported
 - Undefined behaviour, buffer overflows → security risks
 - Weak type system → difficult to reason about correctness, effectively model problem domain

Unix and C

- Unix has proven surprisingly resilient and portable – but is it still the right model?
 - Maybe – work-arounds for its limitations exist:
 - Kernel bypass networking
 - Increasingly baroque package management
 - Containers and sandboxing
 - The C programming language is increasingly a liability
 - Too easy to introduce security vulnerabilities
 - Too easy to trip over undefined behaviour
 - Insufficient abstractions

Some Were Meant for C
The Endurance of an Unmanageable Language

Stephen Kell
Computer Laboratory
University of Cambridge
Cambridge, United Kingdom
stephen.kell@cl.cam.ac.uk

Abstract
The C language leads a double life: as an application programming language of yesteryear, perpetuated by circumstance, and as a systems programming language which remains a weapon of choice decades after its creation. This essay is a C programmer's reaction to the call to abandon ship. It questions several properties commonly held to define the experience of using C; these include unsafety, undefined behaviour, and the motivation of performance. It argues all these are in fact inessential; rather, it traces C's ultimate strength to a *communicative* design which does not fit easily within the usual conception of "a programming language", but can be seen as a counterpoint to so-called "managed languages". This communicativity is what facilitates the essential aspect of system-building: creating parts which interact with other, remote parts—being "alongside" not "within".

CCS Concepts • Software and its engineering → General programming languages; Compilers; • Social and professional topics → History of programming languages;

Keywords systems programming, virtual machine, managed languages, safety, undefined behavior

ACM Reference Format:
Stephen Kell. 2017. Some Were Meant for C. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Vancouver, Canada, October 25–27, 2017 (Onward'17)*, 18 pages.
<https://doi.org/10.1145/3133850.3133867>

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Onward'17, October 25–27, 2017, Vancouver, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5530-8/17/10...\$15.00
<https://doi.org/10.1145/3133850.3133867>

S. Kell, "Some were meant for C: The endurance of an unmanageable language", International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Vancouver, BC, Canada, October 2017. ACM. DOI:[10.1145/3133850.3133867](https://doi.org/10.1145/3133850.3133867)

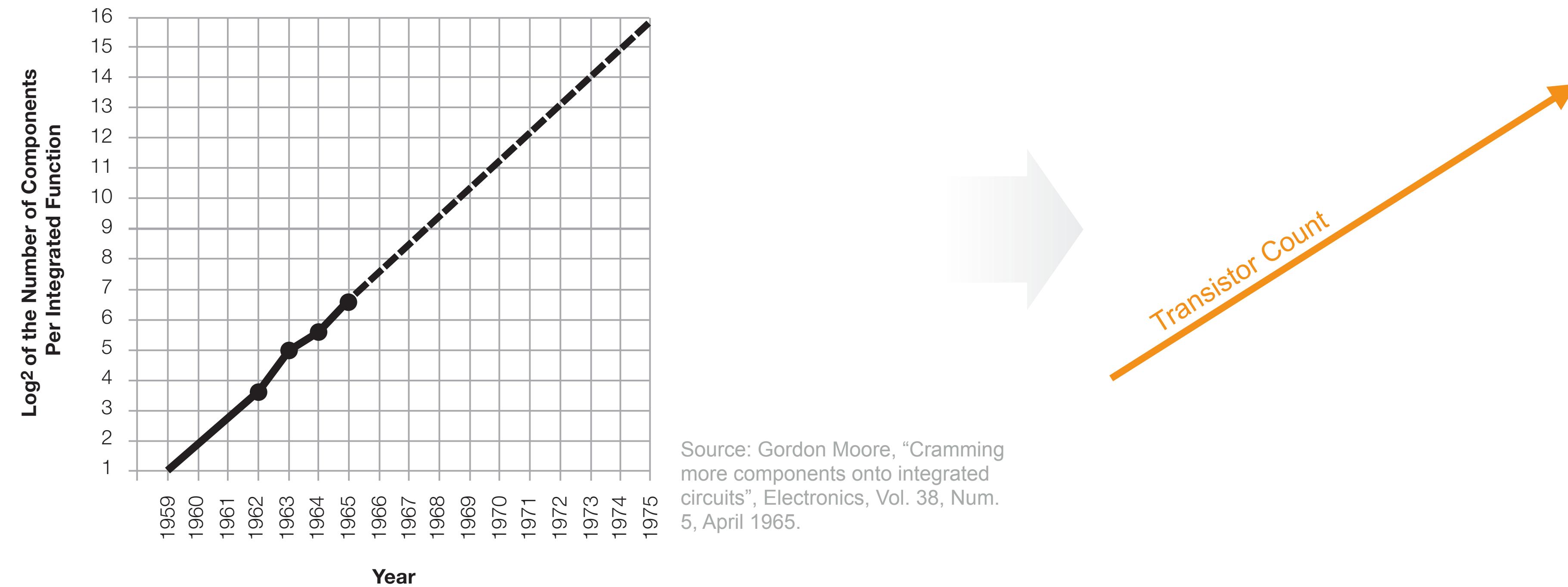
Systems Programming

- What is systems programming?
- **The state of the art**
- Challenges and limitations
- Next steps in systems programming

Challenges and Limitations

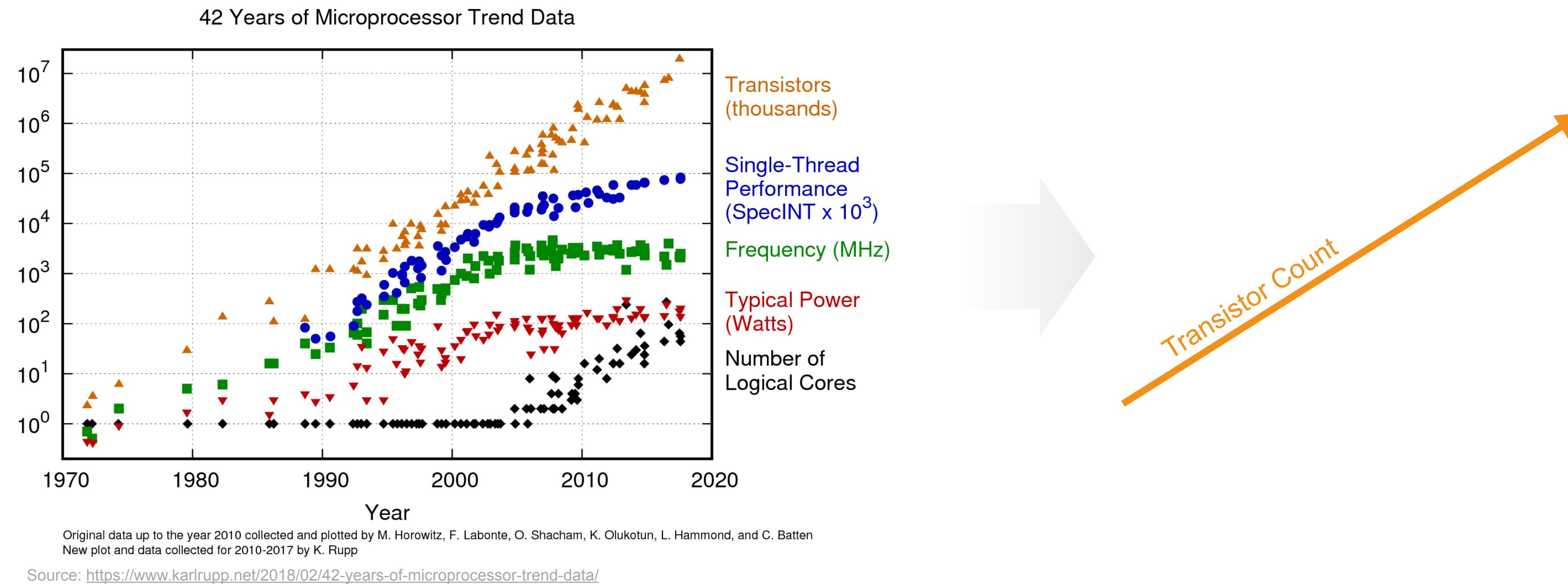
- What changes in the environment are affecting systems programs?
 - The end of Moore's law
 - Increasing concurrency – imposed due to hardware changes
 - Increasing need for security – the Internet
 - Increasing mobility and connectivity

The End of Moore's Law (1/2): Physical Limits



- Moore's law: advances in manufacturing double transistor count every two years

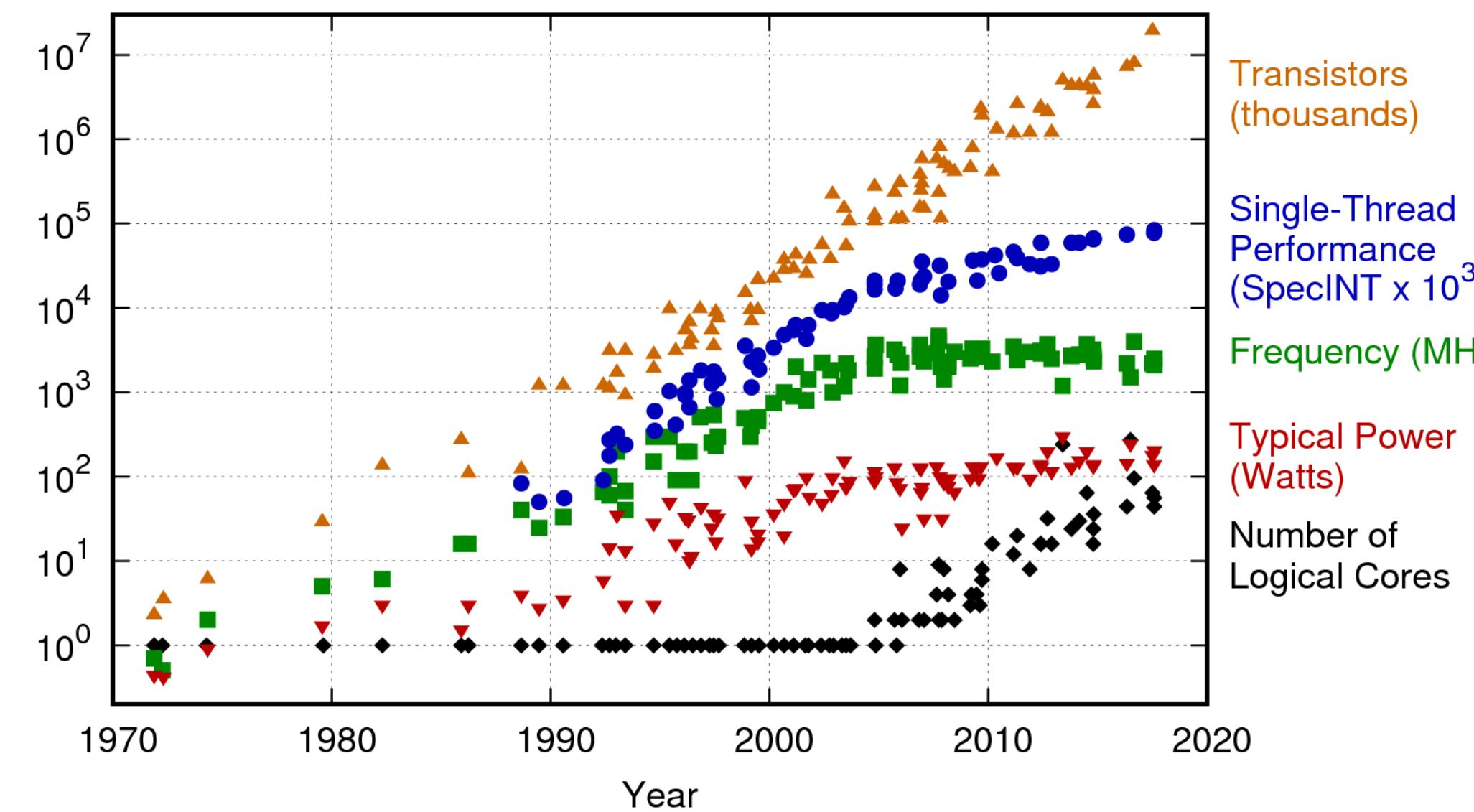
The End of Moore's Law (1/2): Physical Limits



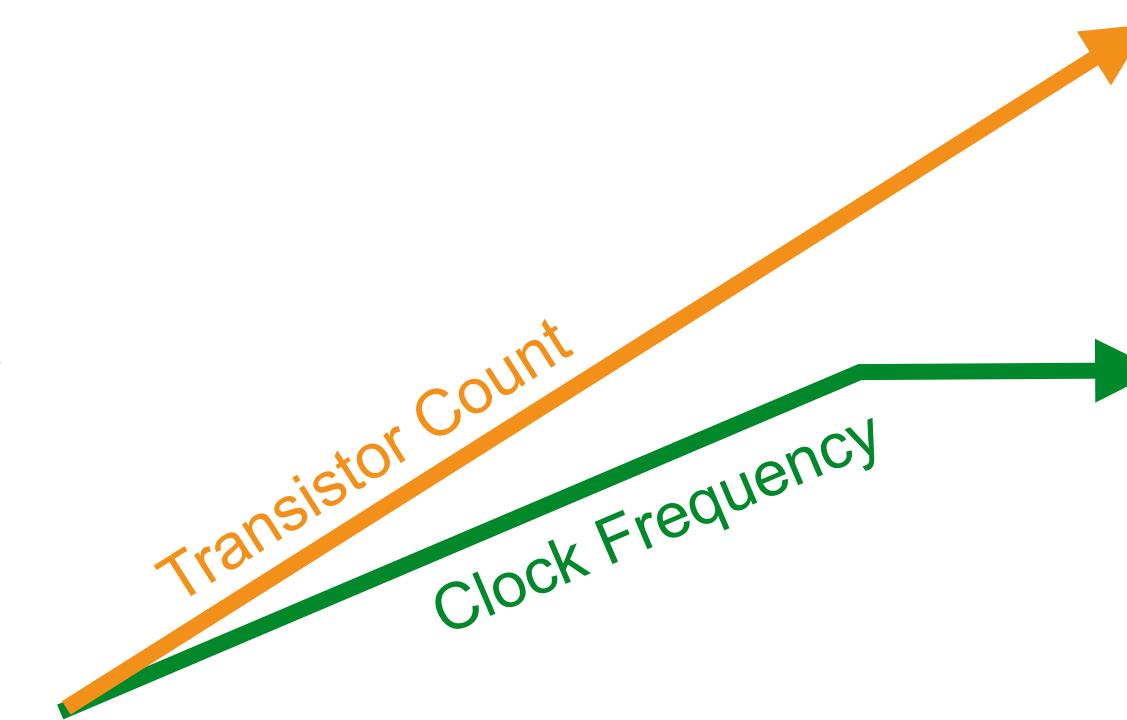
- Moore's law: advances in manufacturing double transistor count every two years
- But, rapidly approaching physical limits:
 - 10nm process → features ~40 atoms across
 - Transistors *will* stop shrinking soon

The End of Moore's Law (2/2): Dennard Scaling

42 Years of Microprocessor Trend Data



Source: <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>

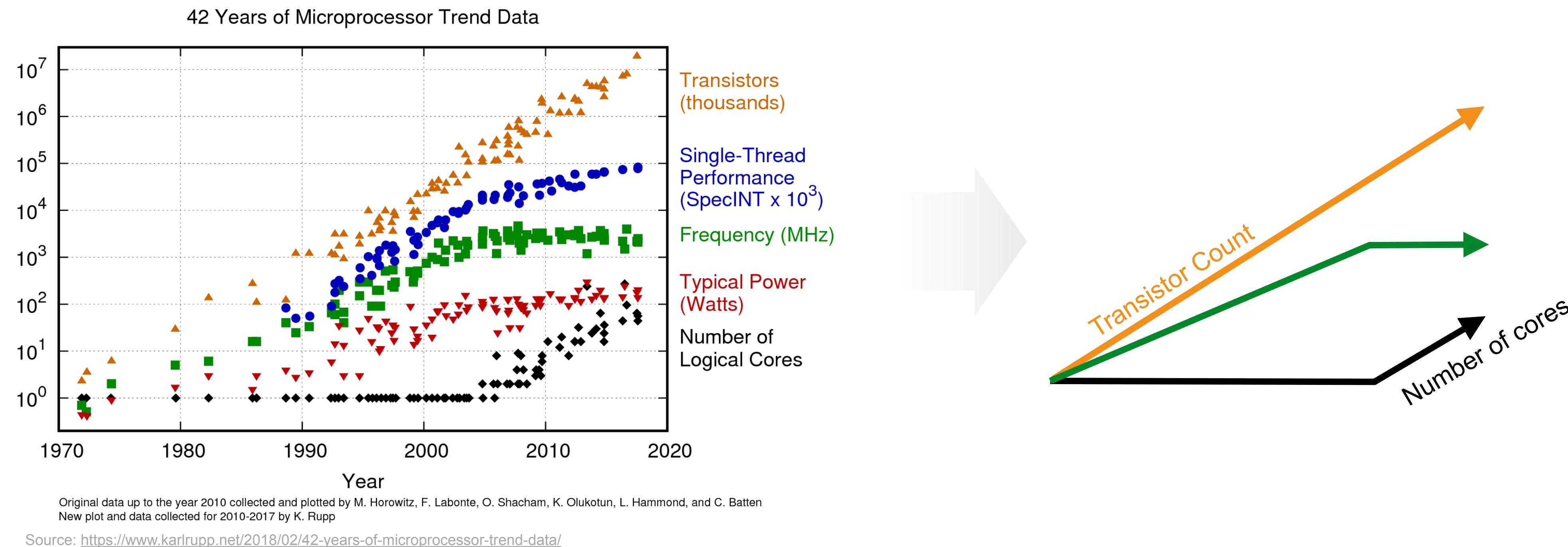


$$\text{Power consumption} \propto C \cdot F \cdot V^2 + L$$

- C = capacitance (transistor size)
- F = frequency (clock rate)
- V = voltage
- L = leakage

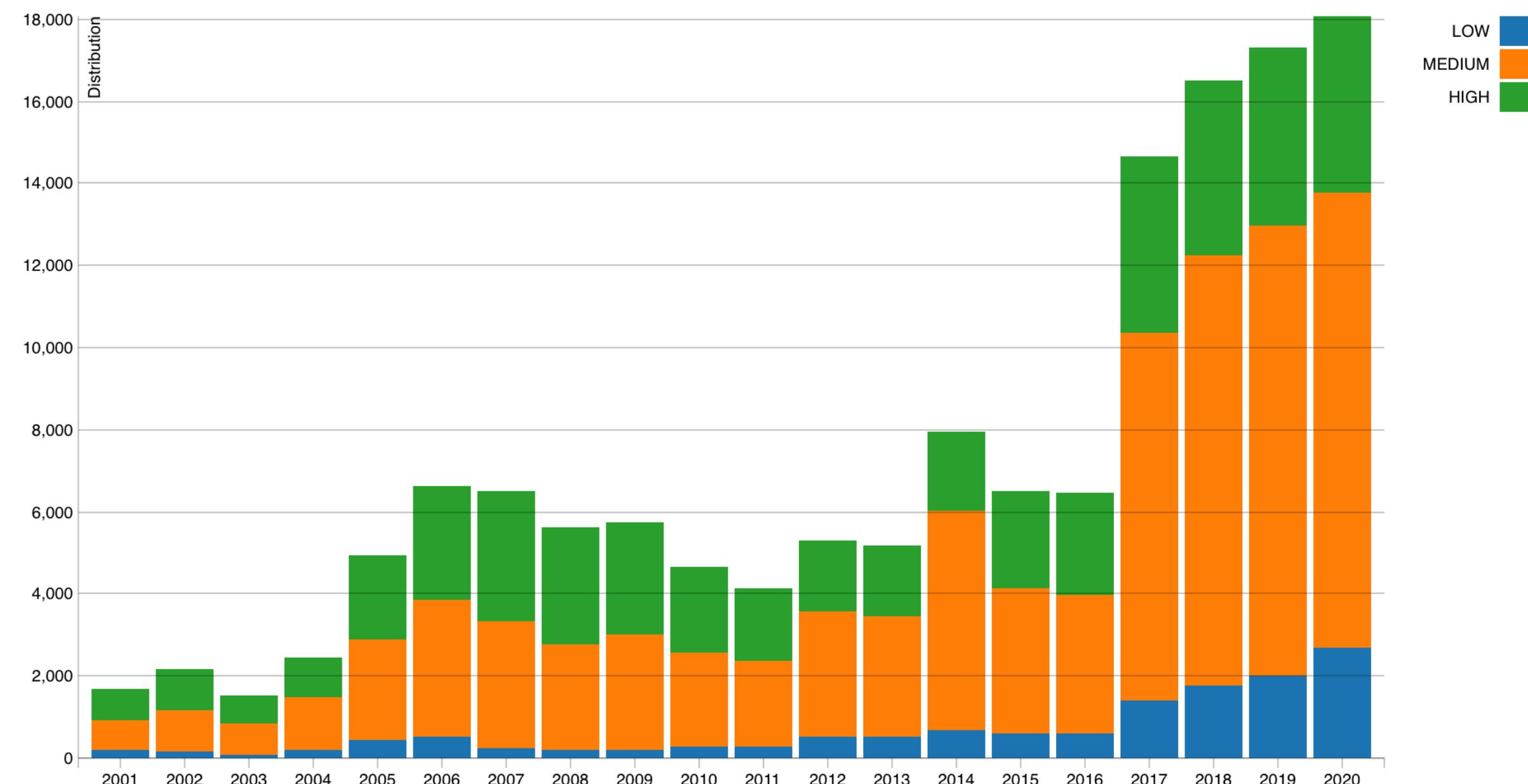
- Dennard scaling: smaller transistors reduce capacitance and voltage: frequency increase without increasing power consumption
- Scaling relation breaks down eventually, due to leakage, and clock frequency increase stalls

Increasing Concurrency



- Breakdown of Dennard scaling limits clock frequency, but Moore's law gives more transistors → used to increase number of cores
- Concurrency related problems become more severe:
 - Performance with correctness, avoiding race conditions and deadlocks

Increasing Need for Security



- Weaponisation of the Internet
- The combination of C and Unix has not proven easy to secure
- **Most** vulnerabilities due to weak typing and lack of memory safety

Source: <https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time>

Number of security vulnerabilities reported per year – US National Vulnerability Database

Increasing Mobility and Connectivity

- Computing increasingly implies mobile devices
 - Always on – but constrained by limitations of battery power
 - Always connected – by increasingly heterogenous networks
- Do we have the APIs, tools, and programming models to make effective use of such devices?

Systems Programming

- What is systems programming?
- The state of the art
- **Challenges and limitations**
- Next steps in systems programming

Next Steps in Systems Programming

- Can strong type systems improve the expressivity, correctness, and security of systems programs?

Next Steps in Systems Programming

- Advances in programming language design are starting to provide the necessary tools – and are beginning to be applied to systems languages
 - Modern type systems
 - Functional programming techniques
- These can help to:
 - Improve memory management and safety – while maintaining control over allocation and data representation
 - Improve security – eliminates common classes of vulnerability
 - Improve support for concurrency – eliminates race conditions
 - Improve correctness – eliminates common classes of bug

What is a Modern Type System? (1/2)

- A modern type system is expressive enough to:
 - Provide useful guarantees about program behaviour
 - Prevent buffer overflows, use-after-free bugs, race conditions, iterator invalidation, ...
 - Provide a model of the problem that prevents inconsistencies in the solution, while avoiding run-time overheads
 - No cost abstractions – compile-time checking that has no run-time cost
 - Describe constraints on program behaviour in the types – the compiler as a debugger

What is a Modern Type System? (2/2)

ACCEPT(2) BSD System Calls Manual ACCEPT(2)

NAME
accept -- accept a connection on a socket

SYNOPSIS

```
#include <sys/socket.h>

int
accept(int socket, struct sockaddr *restrict address,
socklen_t *restrict address_len);
```

DESCRIPTION

The argument socket is a socket that has been created with `socket(2)`, bound to an address with `bind(2)`, and is listening for connections after a `listen(2)`. `accept()` extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of socket, and allocates a new file descriptor for the socket. If no pending connection is present, it blocks.

- Common bug in networking code: call `read()` on listening socket, not socket returned from `accept()` that represents the connection
 - Both file descriptors are represented as type `int`
 - compiler can't check for such misuse
 - If listening socket and connected socket were separate types, and `read()` took a connected socket as its parameter, the bug would be found at compile-time
- Trivial example of important principle – try to describe behaviour in types so compiler can detect logic errors

What is Functional Programming? (1/2)

- A programming style that highlights:
 - Pure, referentially transparent, functions
 - No side effects; no shared mutable state, control over I/O with language support for functions as first class types
- Compare with imperative languages, such as C:
 - Frequent use of shared mutable state, side effects, and impure functions
 - No control over I/O operations
 - Limited abstraction

What is Functional Programming? (2/2)

- Pure functional languages constrain programs
- Haskell is a testbed for exploring pure functional programming
 - Principled, but perhaps impractical for large-scale systems programs
 - Unsued to some programs, natural for others
- But, **concepts are widely applicable**
 - Pure functional code is easy to test and debug – no hidden state
 - Pure functional code is thread safe – no side effects or mutable state
 - Eliminating shared mutable state and controlling I/O avoids race conditions
- Use functional programming ideas where they make sense – prevent certain classes of bugs

How to Improve Memory Management & Safety? (1/2)

- C has manual memory management
 - Pointers, `malloc()` and `free()`
 - Arrays represented as pointers to their first element; don't store length
 - Access outside allocated memory “undefined” but no checks to prevent
- Good reasons for this at the time:
 - Slow machines with limited memory; bounds checks and garbage collection too expensive
 - Not all of these are still valid

```
#include <stdio.h>

int main()
{
    int x[5];

    x[3] = 42;

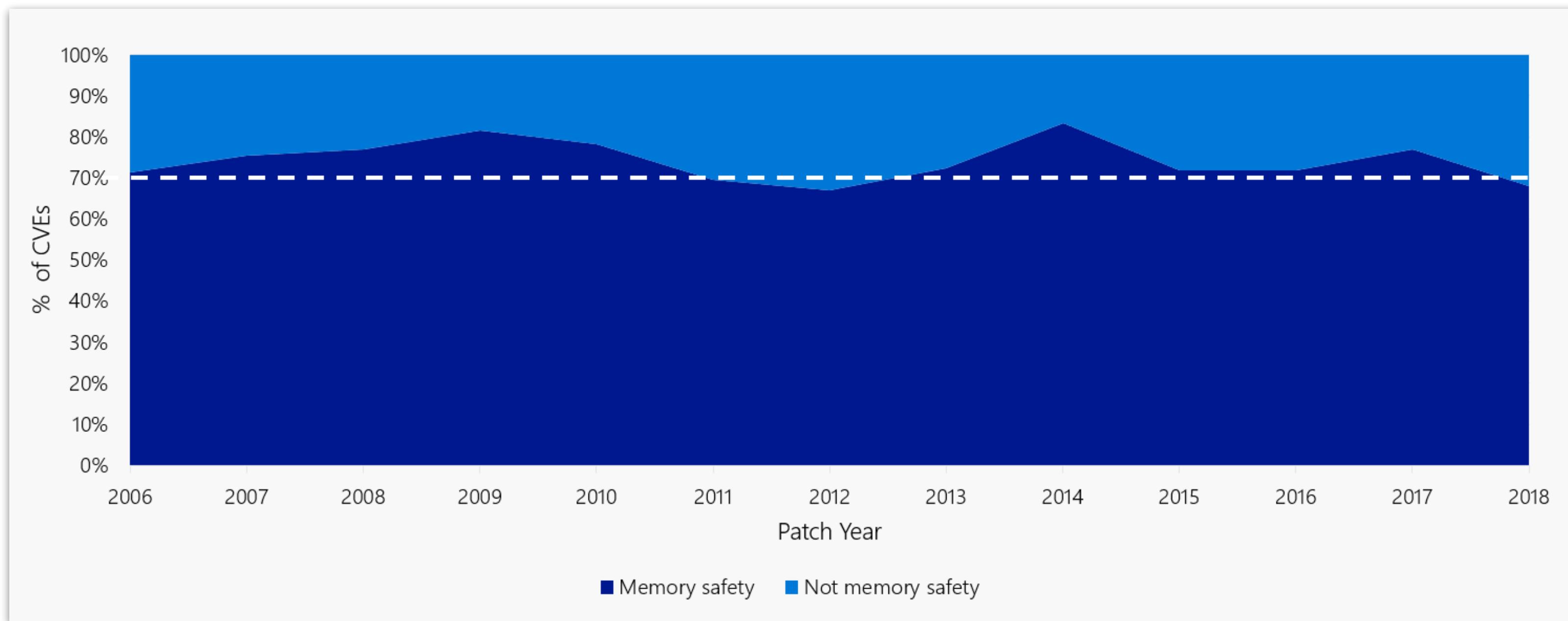
    int a = *(x + 3);
    printf("%d\n", a);

    int b = 3[x];
    printf("%d\n", b);
}
```

How to Improve Memory Management & Safety? (2/2)

- Manual memory management leads to bugs:
 - Use after free
 - Memory leaks
 - Buffer overflows
 - Iterator invalidation
- Modern type systems eliminate these **classes** of bug
 - Enforce bounds checks
 - Enforce ownership of data – code that tries to use data after it's been freed won't compile; similar for iterator invalidation

How to Improve Security?

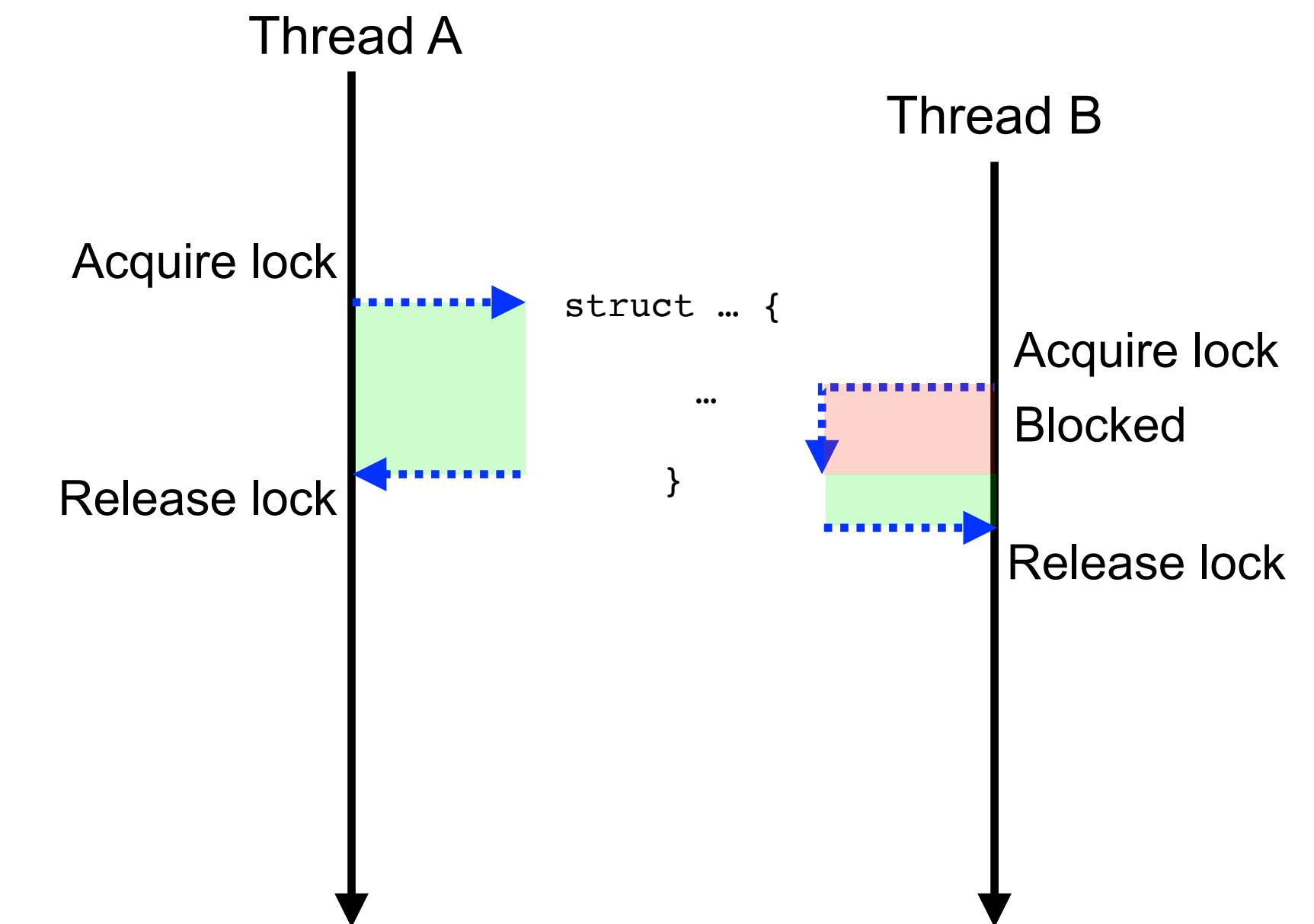


Source: <https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/>

- ~70% of all reported security vulnerabilities are memory safety violations that should be caught by a modern type system – buffer overflows, use-after-free, memory corruption, treating data as executable code
- Use of type based modelling of the problem domain can help address others – by more rigorous checking of assumptions

How to Improve Support for Concurrency?

- Pervasively multicore hardware → concurrent software
 - Common abstractions: threads, locks, shared mutable state
 - Prone to race conditions:
 - Too many or too few locks held
 - Locks held at the wrong time
 - Locks don't compose
- Two approaches to addressing race conditions:
 - Avoid races by avoiding shared mutable state: functional programming, avoiding mutable data
 - Avoid races by requiring single ownership of data objects: message passing, rather than sharing



How to Improve Correctness? (1/2)

- Modern systems programming languages can eliminate certain classes of bug that are common in C
 - Use-after-free, memory leaks, buffer overflows, iterator invalidation
 - Data races in multi-threaded code
- **Don't fix the bug – eliminate the class of bugs**

How to Improve Correctness? (2/2)

- Modern type systems allow for better modelling of the problem domain, and so more checking of code for consistency
- Define types representing the problem domain, rather than using generic types – e.g., if you pass around **PersonId** and **VehicleId** rather than **int**, the compiler will warn if you pass the wrong type of identifier to a function
- Represent program states in the types – e.g., **ListeningSocket** vs. **ConnectedSocket**
- Modern languages allow you to define types and abstractions easily and without run-time cost – type-first design allows code that is correct by construction
- **Use the compiler to debug your design**

Next Steps in Systems Programming

- People can't manage the complexity – need better tooling to help
- C and Unix solve many systems programming problems – control over data representation, memory management, sharing of state
- Emerging, strongly-typed, languages and systems give the same degree of control – with added safety
 - Types help model the problem domain, structure code
 - Types and associated tooling help detect logic errors early – correct by construction
 - We will explore these ideas using the Rust programming language

Summary

- What is systems programming?
- The state of the art
- Challenges and limitations
- Next steps in systems programming

Types and Systems Programming

Advanced Systems Programming (H/M)
Lecture 3



Lecture Outline

- Strongly Typed Languages
 - What is a strongly typed language?
 - Why is strong typing desirable?
 - Types for systems programming
- Introducing the Rust programming language
 - Basic operations and types
 - Pattern matching
 - Memory management
 - Why is Rust interesting?

Strongly Typed Languages

- What is a strongly typed language?
- Why is strong typing desirable?
- Types for systems programming

What is a Type?

- A type describes *what* an item of data represents
 - Is it an integer? floating point value? file? sequence number? username?
 - What, conceptually, is the data?
 - How is it represented?
 - Types are very familiar in programming

```
int    x;
double y;
char  *hello = "Hello, world";

struct sockaddr_in {
    uint8_t      sin_len;
    sa_family_t   sin_family;
    in_port_t     sin_port;
    struct in_addr sin_addr;
    char          sin_pad[16];
};
```

Declaring variables and specifying their type

Declaring a new type,
`struct sockaddr_in`

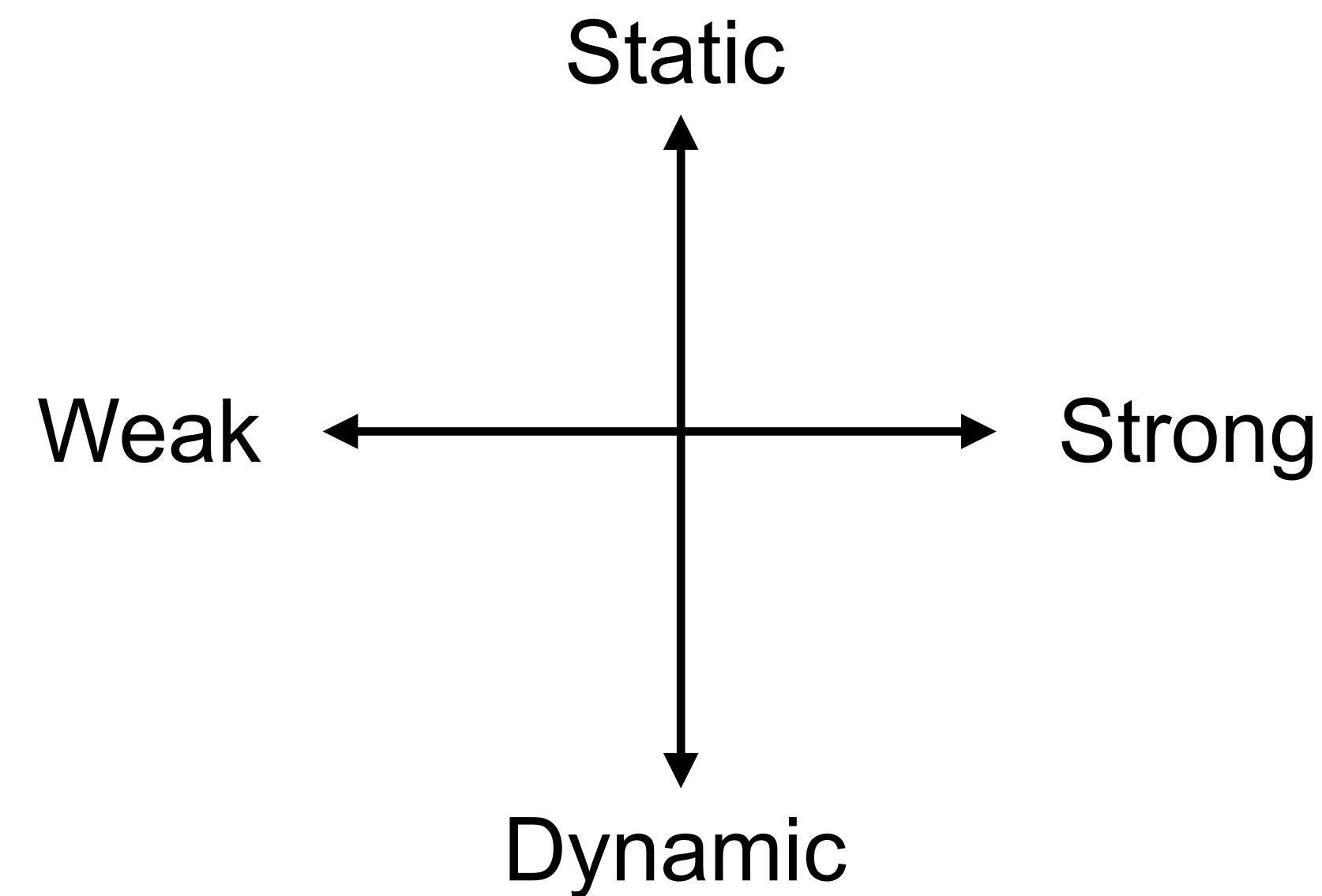
What is a Type System? (1/2)

- A type system is a set of rules constraining how types can be used
 - What operations can be performed on a type?
 - What operations can be performed with a type?
 - How does a type compose with other types of data?

What is a Type System? (2/2)

- A type system proves the absence of certain program behaviours
 - It doesn't guarantee the program is correct
 - It does guarantee that *some* incorrect behaviours do not occur
 - A good type system eliminates common classes of bug, without adding too much complexity
 - A bad type system adds complexity to the language, but doesn't prevent many bugs
 - Type-related checks can happen at compile time, at run time, or both
 - e.g., array bounds checks are a property of an array type, checked at run time

Types of Type Systems



- Can objects change their type?
- How strictly are the typing rules enforced?

Static and Dynamic Types (1/3)

- In a language with static types, the type of a variable is fixed:
 - Some require types to be explicitly declared; others can infer types from context
 - Just because the language can infer the type does not mean the type is dynamic:

```
> cat src/main.rs
fn main() {
    let x = 6;
    x += 4.2;
    println!("{}", x);
}
> cargo build
Compiling hello v0.1.0 (/Users/csp/tmp/hello)
error[E0277]: cannot add-assign `<float>` to `<integer>`
--> src/main.rs:3:7
   |
3 |     x += 4.2;
   |     ^^^ no implementation for `<integer> +<float>`
   |
   = help: the trait `std::ops::AddAssign<<float>>` is not implemented for `<integer>`

error: aborting due to previous error
```

The Rust compiler infers that `x` is an integer and won't let us add a floating point value to it, since that would require changing its type

Static and Dynamic Types (2/3)

- In a language with dynamic types, the type of a variable can change:

```
> python3
Python 3.6.2 (v3.6.2:5fd33b5926, Jul 16 2017, 20:11:06)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> x = 6
>>> type(x)
<class 'int'>
>>> x += 4.2
>>> type(x)
<class 'float'>
>>>
```

Static and Dynamic Types (3/3)

- Dynamically typed languages tend to be lower performance, but offer more flexibility
 - They have to store the type as well as its value, which takes additional memory
 - They can make fewer optimisation based on the type of a variable, since that type can change
- Systems languages generally have static types, and be compiled ahead of time, since they tend to be performance sensitive

Strong and Weak Types (1/2)

- In a language with **strong** types, every operation must conform to the type system
 - Operations that cannot be proved to conform to the typing rules are not permitted
- **Weakly typed** languages provide ways of circumventing the type checker:
 - This might be automatic safe conversions between types:

```
float  x = 6.0;
double y = 5.0;
double z = x + y;
```

- Or it might be an open-ended cast:

```
char *buffer[BUFSIZE];
int   fd = socket(...);
...
if (recv(fd, buffer, BUFSIZE, 0) > 0) {
    struct rtp_packet *p = (struct rtp_packet *) buf;
    ...
}
```

Common C programming idiom: casting between types using pointers to evade the type system

Strong and Weak Types (2/2)

- Think of **strong** and **weak** types in the context of **safe** and **unsafe** languages:
 - A **safe** language, whether static or dynamic, know the types of all variables and only allows legal operations on those values
 - An **unsafe** language allows the types to be circumvented to perform operations the programmer believes correct, but the type system can't prove to be so

Why is Strong Typing Desirable?

- Results of a program using only **strong types** are well-defined → a **safe** language
 - Type system ensures results are consistent with the rules of the language
 - A strongly-typed program will only ever perform operations on a type that are legal – cannot perform undefined behaviour
- Use of **strong types** helps model the problem, check for consistency, and eliminate common classes of bug
- Strong typing **cannot** prove that a program is correct, but **can** prove the absence of certain classes of bug

Segmentation fault (core dumped)

Segmentation faults should never happen:

- Compiler and runtime should strongly enforce type rules
- If program violates them, it should be terminated cleanly
- Security vulnerabilities come from undefined behaviour after type violations

Segmentation fault (core dumped)

3.4.3

1 undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

2 NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

C has 193 kinds of undefined behaviour

Appendix J of the C standard <https://www.iso.org/standard/74528.html> (\$) or http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf

Undefined behaviour leads to *entirely unpredictable* results → <https://blog.regehr.org/archives/213>

Types for Systems Programming

- C is weakly typed and widely used for systems programming
 - Why is this?
 - Can systems programming languages be strongly typed?
 - What are the challenges in strongly typed systems programming?

Why is C Weakly Typed?

- Mostly, historical reasons:
 - The original designers of C were not type theorists
 - The original machines on which C was developed didn't have the resources to perform complex type checks
 - Type theory was not particularly advanced in the early 1970s

Is Strongly-typed Systems Programming Feasible?

- Yes – many examples of operating systems written in strongly-typed languages
 - Old versions of macOS written in Pascal
 - Project Oberon <http://www.projectoberon.com>
 - US DoD and the Ada programming language
 - Aerospace, military, air traffic control
 - Popularity of Unix and C has led to a belief that operating systems require unsafe code
 - True only at the very lowest levels
 - Most systems code, including device drivers, can be written in strongly typed, safe, languages
 - **Rust is a modern attempt to provide a type-safe language suited to systems programming**

```
type ErrorType    is range 0..15;
type UnitSelType is range 0..7;
type ResType      is range 0..7;
type DevFunc      is range 0..3;
type Flag         is (Set, NotSet);
type ControlRegister is
record
    errors      : ErrorType;
    busy        : Flag;
    unitSel    : UnitSelType;
    done        : Flag;
    irqEnable  : Flag;
    reserved   : ResType;
    devFunc    : DevFunc;
    devEnable  : Flag;
end record;

for ControlRegister use
record
    errors      at 0*Word range 12..15;
    busy        at 0*Word range 11..11;
    unitSel    at 0*Word range 8..10;
    done        at 0*Word range 7.. 7;
    irqEnable  at 0*Word range 6.. 6;
    reserved   at 0*Word range 3.. 5;
    devFunc    at 0*word range 1.. 2;
    devEnable  at 0*Word range 0.. 0;
end record;

for ControlRegister'Size use 16;
for ControlRegister'Alignment use Word;
for ControlRegister'Bit_order use Low_Order_First;
...
```

Strongly Typed Languages

- What is a strongly typed language?
- Why is strong typing desirable?
- Types for systems programming

Introducing Rust

- What is Rust?
 - **Basic operations and types**
 - Abstraction: Traits, Enumerated types and pattern matching
 - Memory allocation and boxes
- Why is it interesting?

The Rust Programming Language

- A modern systems programming language with a strong static type system
- Initially developed by Graydon Hoare as a side project, starting 2006
- Sponsored by Mozilla since 2009
- Rust v1.0 released in 2015
- Rust v1.31 “Rust 2018 Edition” released December 2018
 - Backwards compatible – but tidies up the language
 - <https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html>
- New releases made every six weeks – strong backwards compatibility policy



Basic Features and Types (1/10)

```
fn main() {  
    println!("Hello, world!");  
}
```

Function definition: `main()` takes no arguments, returns nothing

Macro expansion: `println!()` with string literal as argument

Basic Features and Types (2/10)

```
use std::env;

fn main() {
    for arg in env::args() {
        println!("{}:?", arg);
    }
}
```

Import **env** module from standard library

Use **for** loop to iterate over command line arguments

```
$ rustc args.rs
$ ./args 1 2 3
"./args"
"1"
"2"
"3"
$
```

Basic Features and Types (3/10)

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m!=0 {
        if m < n {
            let t = m;
            m = n;
            n = t;
        }
        m = m % n;
    }
    n
}

fn main() {
    let m = 12;
    let n = 16;
    let r = gcd(m, n);
    println!("gcd({}, {}) = {}", m, n, r);
}
```

\$ rustc gcd.rs
\$./gcd
gcd(12, 16) = 4
\$

Function arguments and return type; mutable vs immutable

Control flow: **while** and **if** statements

Local variable definition (**let** binding); type is inferred

Function implicitly returns value of final expression
(**return** statement allows early return)

Function call, assigning result to local variable

Basic Features and Types (4/10)

C	Rust
<code>unsigned</code>	<code>usize</code>
<code>uint8_t, unsigned char</code>	<code>u8</code>
<code>uint16_t</code>	<code>u16</code>
<code>uint32_t</code>	<code>u32</code>
<code>uint64_t</code>	<code>u64</code>
C	Rust
<code>int</code>	<code>isize</code>
<code>int8_t, signed char</code>	<code>i8</code>
<code>int16_t</code>	<code>i16</code>
<code>int32_t</code>	<code>i32</code>
<code>int64_t</code>	<code>i64</code>
<code>float</code>	<code>f32</code>
<code>double</code>	<code>f64</code>
<code>int</code>	<code>bool</code>
	<code>char</code>

- Primitive types map closely to those in C
 - Rust has native `bool`, C uses `int` to represent boolean
 - In C, a `char` is one byte, implementation defined if signed, character set unspecified
 - In Rust, a `char` is a 32-bit Unicode scalar value:
 - Unicode scalar value ≠ code point ≠ grapheme cluster ≠ “character”
 - e.g., ü is two scalar values “Latin small letter U (U+0075)” + “combining diaeresis (U+0308)”, but one grapheme cluster (<https://crates.io/crates/unicode-segmentation> – text is hard)

<https://doc.rust-lang.org/book/ch03-02-data-types.html>

Basic Features and Types (5/10)

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
    let b = a[2];  
    println!("b={}", b);  
}
```

Arrays work as expected; elements are all the same type; number of elements fixed
Type of the array is inferred from type of elements
Array types are written `[T]`, where `T` is the type of the elements

```
$ rustc array.rs  
$ ./args  
b=3  
$
```

Basic Features and Types (6/10)

```
fn main() {  
    let mut v : Vec<u32> = Vec::new();  
    v.push(1);  
    v.push(2);  
    v.push(3);  
    v.push(4);  
    v.push(5);  
}
```

Vectors, `Vec<T>`, are the variable sized equivalent of arrays
The type parameter `<T>` specifies element

The `: Vec<u32>` modifier specifies the type for variable `v`
and can usually be omitted, allowing compiler to infer type:

```
let mut v = Vec::<u32>::new();
```

```
fn main() {  
    let v = vec![1, 2, 3, 4, 5];  
}
```

The `vec![...]` macro is a shortcut to create vector literals

Vectors implemented as the equivalent of a C program that uses `malloc()` to allocate
space for an array, then `realloc()` to grow the space when it gets close to full.

A vector, `Vec<T>`, can be passed to a function that expects a reference to an array `[T]`
(`Vec<T>` implements the trait `Deref<Target=&[T]>` that defines the conversion)

Basic Features and Types (7/10)

```
fn main() {  
    let tup = (500, 6.4, 1);  
  
    let (x, y, z) = tup;  
  
    println!("The value of y is: {}", y);  
    println!("The 2nd element is {}", tup.1)  
}
```

()

Tuples are collections of unnamed values; each element can be a different type

let bindings can de-structure tuples

Tuple elements can be accessed by index

An empty tuple is the unit type (like **void** in C)

Basic Features and Types (8/10)

```
struct Rectangle {  
    width: u32,  
    height: u32  
}  
  
fn area(rectangle: Rectangle) -> u32 {  
    rectangle.width * rectangle.height  
}  
  
fn main() {  
    let rect = Rectangle { width: 30, height: 50 };  
  
    println!("Area of rectangle is {}", area(rect));  
}
```

Structs are collections of named values;
each element can have a different type
<https://doc.rust-lang.org/book/ch05-00-structs.html>

Access fields in struct using dot notation

Creates a struct, specifying field values

```
$ rustc struct.rs  
$ ./struct  
Area of rectangle is 1500  
$
```

Basic Features and Types (9/10)

```
struct Point(i32, i32, i32);  
  
let origin = Point(0, 0, 0);
```

Elements of a struct can be unnamed: known as tuple structs
useful as type aliases

```
struct Marker;
```

Unit-like structs have no elements and take up no space
useful as markers or type parameters

Basic Features and Types (10/10)

```
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
impl Rectangle {  
    fn area(self) -> u32 {  
        self.width * self.height  
    }  
}  
  
fn main() {  
    let rect = Rectangle { width: 30, height: 50 };  
  
    println!("Area of rectangle is {}", rect.area());  
}
```

Methods defined in `impl` block

Explicit `self` references, like Python

Method call uses dot notation

Methods can be implemented on structs – somewhat similar to objects, but Rust does not support inheritance or sub-types

Introducing Rust

- What is Rust?
 - **Basic operations and types**
 - Abstraction: Traits, Enumerated types and pattern matching
 - Memory allocation and boxes
- Why is it interesting?

Introducing Rust

- What is Rust?
 - Basic operations and types
 - **Abstraction: Traits, Enumerated types and pattern matching**
 - Memory allocation and boxes
- Why is it interesting?

Traits (1/5)

```
trait Area {  
    fn area(self) -> u32;  
}  
  
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
impl Area for Rectangle {  
    fn area(self) -> u32 {  
        self.width * self.height  
    }  
}
```

Define a trait with a single method that must be implemented

Define a struct type, **Rectangle**

Implement the **Area** trait on the **Rectangle** type

<https://doc.rust-lang.org/book/ch10-02-trait.html>

- Traits describe **functionality** that types can implement
 - Methods that must be provided, and associated types that must be specified, by types that implement the trait – but no instance variables or data
 - Similar to type classes in Haskell or interfaces in Java

Traits (2/5)

```
trait Area {  
    fn area(self) -> u32;  
}  
  
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
impl Area for Rectangle {  
    fn area(self) -> u32 {  
        self.width * self.height  
    }  
}
```

A trait can be implemented by multiple types – here we also implement the **Area** trait for the **Circle** type

```
struct Circle {  
    radius: u32  
}  
  
impl Area for Circle {  
    fn area(self) -> u32 {  
        PI * self.radius * self.radius  
    }  
}
```

- Traits are an important tool for abstraction – similar role to sub-types in many languages

Traits (3/5): Generic Functions

- Rust uses traits instead of classes and inheritance to define generic functions or methods that work with any type that implements a particular trait
- Define a trait:

```
trait Summary {  
    fn summarise(self) -> String;  
}
```

- Write functions that work on types that implement that trait:

```
fn notify<T: Summary>(item: T) {  
    println!("Breaking news! {}", item.summarise());  
}
```

Type parameter in angle brackets: **T** is any type that implement the **Summary** trait

Traits (4/5): Deriving Common Traits

- The `derive` attribute makes compiler automatically generate implementations of some common traits:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

The `#[derive(T)]` annotation makes compiler generate `impl` block with standard implementation of methods for derived trait `T`

- Compiler implements this for traits in the standard library that are always implemented the same way:
 - <https://doc.rust-lang.org/book/appendix-03-derivable-traits.html>
 - Can also be implemented for user-defined traits:
 - Only useful if every implementation of the trait will follow the exact same structure
 - <https://doc.rust-lang.org/book/ch19-06-macros.html#how-to-write-a-custom-derive-macro>

Traits (5/5): Associated Types

- Traits can also specify associated types – types that must be specified when a trait is implemented
- Example: **for** loops operate on iterators

```
fn main() {
    let a = [42, 43, 44, 45, 46];

    for x in a.iter() {
        println!("x={}", x);
    }
}
```

The `a.iter()` function call returns an iterator over the array

- An iterator is something that implements the **Iterator** trait:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    // more...
}
```

An `impl` of **Iterator** must define the type of `item`, as well as implementing the methods

Enumerated Types (1/3)

```
enum TimeUnit {  
    Years, Months, Days, Hours, Minutes, Seconds  
}
```

Basic enums work just like in C

<https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html>

```
enum RoughTime {  
    InThePast(TimeUnit, u32),  
    JustNow,  
    InTheFuture(TimeUnit, u32)  
}  
  
let when = RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);
```

Enums also generalise to store tuple-like variants:

```
enum Shape {  
    Sphere {centre: Point3d, radius: f32},  
    Cuboid {corner1: Point3d, corner2: Point3d}  
}  
  
let unit_sphere = Shape::Sphere{centre: ORIGIN, radius: 1.0};
```

...and struct-like variants:

Enumerated Types (2/3)

- An **enum** is used when a variable, parameter, or result can have one of several possible types
 - An **enum** defines alternative types for a type
 - An **enum** can have type parameters that must be defined when the enum is instantiated:

```
enum Result<T, E> {  
    Ok(T),  
    Err(E)  
}
```

- An **enum** can have methods and can implement traits
- Use **enum** to model data that can take one of a set of related values

Enumerated Types (3/3)

- Standard library has two extremely useful standard `enum` types
- The `Option` type represents optional values
 - In C, one might write a function to lookup a key in a database:

```
value *lookup(struct db* self, key *k) {  
    // ...  
}
```

this returns a pointer to the value, or `null` if the key doesn't exist

- In Rust, the equivalent function would return `Option<Value>`:

```
fn lookup(self, key : Key) -> Option<Value> {  
    // ...  
}
```

- The `result` type similarly encodes success or failure:

```
fn recv(self) -> Result<Message, NetworkError> {  
    // ...  
}
```

Definitions in the standard library:

```
enum Option<T> {  
    Some(T),  
    None  
}
```

```
enum Result<T, E> {  
    Ok(T),  
    Err(E)  
}
```

Pattern Matching (1/4)

- Rust match expressions generalise the C switch statement
- Match against constant expressions and wildcards:

```
match meadow.count_rabbits() {  
    0 => {} // nothing to say  
    1 => println!("A rabbit is nosing around in the clover."),  
    n => println!("There are {} rabbits hopping about in the meadow", n)  
}
```

<https://doc.rust-lang.org/book/ch06-02-match.html>

- The value of **meadow.count_rabbits()** is matched against the alternatives
- If matches the constants 0 or 1, the corresponding branch executes
- If none match, the value is stored in the variable **n** and that branch executes
 - Matching against **_** gives a wildcard without assigning to a variable

Pattern Matching (2/4)

- Patterns can be any type, not just integers

```
let calendar = match settings.get_string("calendar") {  
    "gregorian" => Calendar::Gregorian,  
    "chinese"    => Calendar::Chinese,  
    "ethiopian"  => Calendar::Ethiopian,  
    _              => return parse_error("calendar", other)  
};
```

- The `match` expression evaluates to the value of the chosen branch
 - Allows, e.g., use in `let` bindings, as shown

Pattern Matching (3/4)

- Patterns can match against enum values:

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}

let when = RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);
```

```
match rt {
    RoughTime::InThePast(units, count) => format!("{} {} ago", count, units.plural()),
    RoughTime::JustNow                  => format!("just now"),
    RoughTime::InTheFuture(units, count) => format!("{} {} from now", count, units.plural())
}
```

- Selects from different types of data, expressed as **enum** variants
- Must match against all possible variants, or include a wildcard – else compile error

Patterns Matching (4/4)

- C functions often return pointer to value, or **null** if the value doesn't exist
 - Easy to forget the **null** check when using the value:

```
customer *get_user(struct db *db, char *username) {  
    // ...  
}  
  
customer *c = get_user(db, customer_name);  
book_ticket(c, event);
```

- Program crashes with null pointer dereference at run-time if user is not found
- Rust function return an **Option** and pattern match on result:

```
fn get_user(self, username : String) -> Option<Customer> {  
    // ...  
}  
  
match db.get_user(customer_name) {  
    Some(customer) => book_ticket(customer, event),  
    None           => handle_error()  
}
```

Why is this better? All enum variants must be handled, so won't compile if you forget to check the error case

<https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html#the-option-enum-and-its-advantages-over-null-values>

Introducing Rust

- What is Rust?
 - Basic operations and types
 - **Abstraction: Traits, Enumerated types and pattern matching**
 - Memory allocation and boxes
- Why is it interesting?

Introducing Rust

- What is Rust?
 - Basic operations and types
 - Abstraction: Traits, Enumerated types and pattern matching
 - **Memory allocation and boxes**
- Why is it interesting?

References (1/2)

- References are explicit – like pointers in C
 - Create a variable binding:

```
let x = 10;
```

```
int x = 10;
```

- Take a reference (pointer) to that binding:

```
let r = &x;
```

```
int *r = &x;
```

References (1/2)

- References are explicit – like pointers in C
 - Create a variable binding:

```
let x = 10;
```

```
int x = 10;
```

- Take a reference (pointer) to that binding:

```
let r = &x;
```

```
int *r = &x;
```

- Explicitly dereference to access value:

```
let s = *r
```

```
s = *r;
```

References (1/2)

- References are explicit – like pointers in C
 - Create a variable binding:

```
let x = 10;
```

```
int x = 10;
```

- Take a reference (pointer) to that binding:

```
let r = &x;
```

```
int *r = &x;
```

- Explicitly dereference to access value:

```
let s = *r
```

```
s = *r;
```

- Functions can take parameters by reference:

```
fn calculate_length(b: &Buffer) -> usize {  
    // ...  
}
```

```
size_t calculate_length(buffer *b) {  
    // ...  
}
```

References (2/2)

- Immutable references: &

```
fn main() {  
    let x = 10;  
    let r = &x;  
  
    *r = 15;  
  
    println!("x={}", x);  
}
```

immutable reference – can't be changed

compile error: cannot assign to *r which is behind an & reference

References (2/2)

- Immutable references: &

```
fn main() {  
    let x = 10;  
    let r = &x;  
  
    *r = 15;  
  
    println!("x={}", x);  
}
```

immutable reference – can't be changed

compile error: cannot assign to *r which is behind an & reference

- Mutable references: &mut

```
fn main() {  
    let mut x = 10;  
    let r = &mut x;  
  
    *r = 15;  
  
    println!("x={}", x);  
}
```

mutable reference – referenced value *can* change

Constraints on References

- References can never be `null` – they always point to a valid object
 - `Option<T>` indicates an optional value of type `T` where C would use a potentially null pointer

Constraints on References

- References can never be `null` – they always point to a valid object
- There can be many immutable references (`&`) to an object in scope at once, but there cannot be a mutable reference (`&mut`) to the same object in scope
 - An object becomes immutable while immutable references to it are in scope

Constraints on References

- References can never be `null` – they always point to a valid object
- There can be many immutable references (`&`) to an object in scope at once, but there cannot be a mutable reference (`&mut`) to the same object in scope
- There can be at most *one* mutable reference (`&mut`) to an object in scope and there can be no immutable references (`&`) to the object while the mutable reference exists
 - An object is inaccessible to its owner while the mutable reference exists

Constraints on References

- References can never be `null` – they always point to a valid object
- There can be many immutable references (`&`) to an object in scope at once, but there cannot be a mutable reference (`&mut`) to the same object in scope
- There can be at most *one* mutable reference (`&mut`) to an object in scope and there can be no immutable references (`&`) to the object while the mutable reference exists
- These rules are enforced at compile time and prevent null pointer exceptions, iterator invalidation, data races between threads → Lecture 5

Memory Allocation and Boxes (1/2)

- A **Box<T>** is a smart pointer that refers to memory allocated on the heap:

```
fn box_test() {  
    let b = Box::new(5);  
    println!("b = {}", b);  
}
```

```
void box_test() {  
    int *b = malloc(sizeof(int));  
    *b = 5;  
    printf("b = %d\n", *b);  
    free(b);  
}
```

Memory Allocation and Boxes (1/2)

- A **Box<T>** is a smart pointer that refers to memory allocated on the heap:

```
fn box_test() {  
    let b = Box::new(5);  
    println!("b = {}", b);  
}
```

```
void box_test() {  
    int *b = malloc(sizeof(int));  
    *b = 5;  
    printf("b = %d\n", *b);  
    free(b);  
}
```

- Rust makes guarantees about memory allocation:
 - The value returned by `Box::new()` is guaranteed to be initialised
 - The allocated memory is guaranteed to match the size of the type it is to store
 - Rust guarantees that the memory will be automatically deallocated when the box goes out of scope

Memory Allocation and Boxes (2/2)

- Boxes own and, if bound as `mut`, may change the data they store on the heap

```
fn main() {  
    let mut b = Box::new(5);  
    *b = 6;  
    println!("b = {}", b);  
}
```

Memory Allocation and Boxes (2/2)

- Boxes own and, if bound as `mut`, may change the data they store on the heap

```
fn main() {
    let mut b = Box::new(5);
    *b = 6;
    println!("b = {}", b);
}
```

- Boxes do *not* implement the standard **Copy** trait; can pass boxes around, but only one copy of each box can exist – again, to avoid data races between threads
 - A `Box<T>` is a pointer to the heap allocated memory; if it were possible to copy the box, we could get multiple mutable references to that memory

Strings

- Strings are Unicode text encoded in UTF-8 format
- A **str** is an immutable string slice, always accessed via an **&str** reference

```
let s1 = "Hello, World!";
```

String literals are of type **&str**

- The **&str** type is built-in to the language

Strings

- Strings are Unicode text encoded in UTF-8 format
- A **str** is an immutable string slice, always accessed via an **&str** reference
- A **String** is a mutable string buffer type, implemented in the standard library

```
let s2 = String::new();
s2.push_str("Hello, World");
s2.push('!');
```

```
let s3 = String::from("Hello, World");
s3.push('!');
```

Strings

- Strings are Unicode text encoded in UTF-8 format
- A **str** is an immutable string slice, always accessed via an **&str** reference
- A **String** is a mutable string buffer type, implemented in the standard library

```
let s2 = String::new();
s2.push_str("Hello, World");
s2.push('!');
```

```
let s3 = String::from("Hello, World");
s3.push('!');
```

- The **string** type implements the **Deref<Target=str>** trait, so taking a reference to a **String** results actually returns an **&str**

```
let s = String::from("test");      s has type String
let r = &s;                      r has type &String
let t : &str = &s;                t has type &str
```

- This conversion has zero cost, so functions that don't need to mutate the string tend to be only implemented for **&str** and not on **String** values

Rust – Key Points

- Largely traditional systems programming language: basic types, control flow, and data structures are very familiar
- Key innovations in a systems language:
 - Enumerated types and pattern matching
 - `Option` and `Result`
 - Structure types and traits as an alternative to object oriented programming
 - Multiple reference types and ownership

Rust – Key Points

- Largely traditional systems programming language: basic types, control flow, and data structures are very familiar
- Key innovations in a systems language:
 - Enumerated types and pattern matching
 - `option` and `Result`
 - Structure types and traits as an alternative to object oriented programming
 - Multiple reference types and ownership
- **Little in Rust is novel**
 - Rust adopts ideas from research languages:
 - Syntax is a mixture of C and Standard ML
 - Basic data types are heavily influenced by C and C++
 - Enumerated types and pattern matching adapted from Standard ML
 - Traits adapted from Haskell type classes
 - Many influences from C++, but generally “see how C++ does it, and do the opposite”
 - References and ownership rules extend ideas originally developed in Cyclone – this is where Rust has new ideas

Why is Rust interesting? (1/3)

- A modern type system and runtime
 - No concept of undefined behaviour
 - Memory safe
 - No buffer overflows
 - No dangling pointers
 - No null pointer dereferences
 - Zero cost abstractions to model problem space and check consistency of design → lecture 4

Why is Rust interesting? (2/3)

- A type system that can model data and resource ownership
 - Deterministic automatic memory management → lectures 5 and 6
 - Prevents iterator invalidation
 - Prevents use-after-free bugs
 - Prevents most memory leaks
 - Rules around references and ownership prevent data races in concurrent code → lecture 7
 - Enforces the design patterns common in well-written C programs

Why is Rust interesting? (3/3)

- A systems programming language that eliminates many **classes** of bug that are common in C and C++ programs

Summary

- What is a strongly typed language?
- Why is strong typing desirable?
- Types for systems programming
- Introduction to Rust



University
ofGlasgow

Type-based Modelling and Design

Advanced Systems Programming (H/M)
Lecture 4



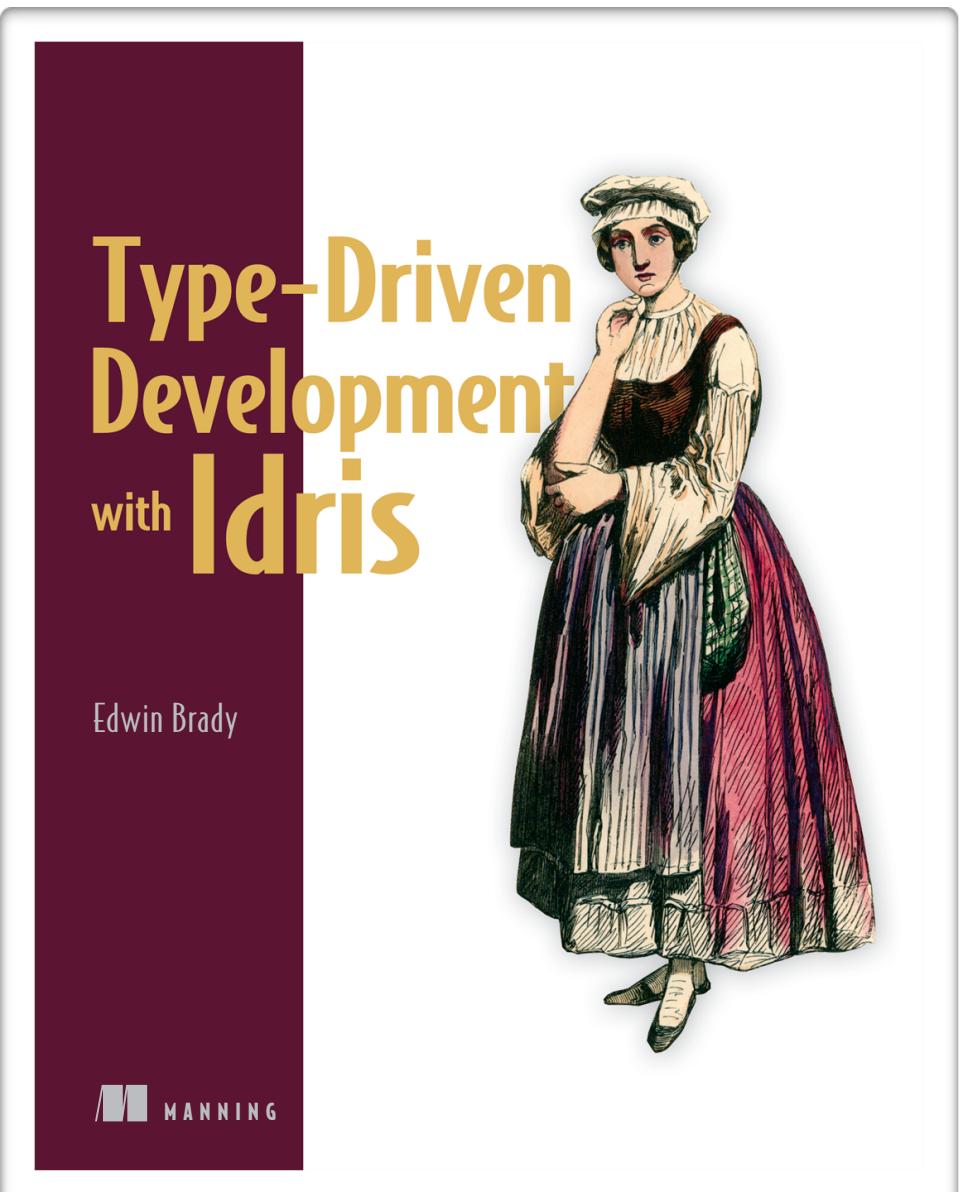
Colin Perkins | <https://csperrkins.org/> | Copyright © 2020 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License.
To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Type-driven Development

- Define the types
- Write the functions
- Refine as needed

Type-driven Development

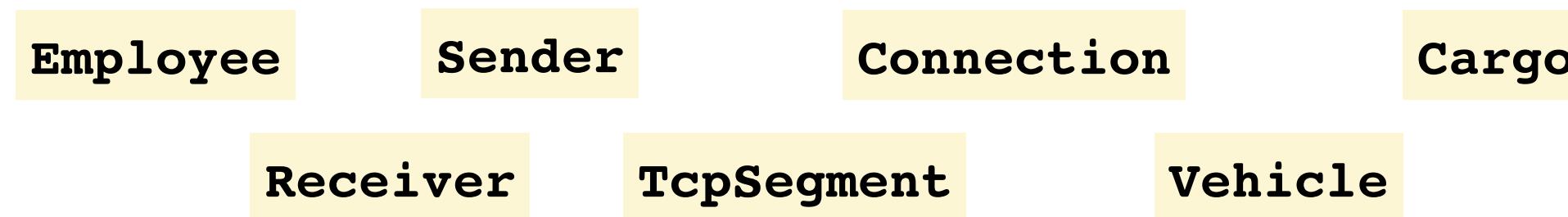
- With expressive, strongly typed, languages – such as Rust, Swift, OCaml – can use the type system to help ensure correctness
- **Define the types first**
- Using the types as a guide, **write the functions**
 - Write the input and output types
 - Write the function, using the structure of the types as a guide
- **Refine** and edit types and functions as necessary
- Don't think of the types as checking the code, think of them as a plan – a model – for the solution



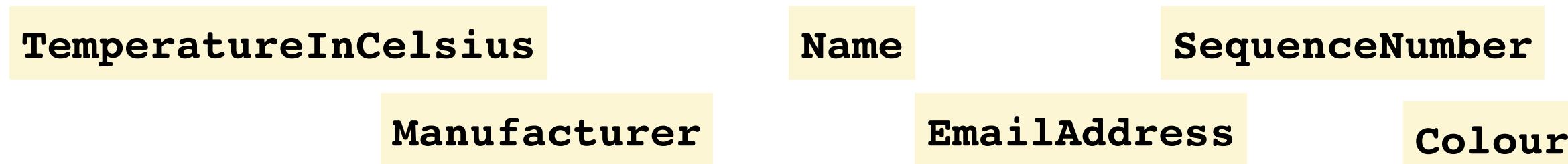
Type-drive development approach adapted from: E. Brady, "[Type-Driven Development with Idris](#)", Manning, March 2017.

Define the Types (1/2)

- Define the types needed to build a domain model
 - Who is interacting? What do they interact with? What sorts of things do they exchange?



- What sort of properties describe those people and things? What data is associated with each?



- What states can the interaction be in?



- Types might be ill-defined and abstract – write them down anyway and refine later

Define the Types (2/2)

- Associate properties with the types:
 - What data is associated with a thing? What properties does it have?

```
struct Sender {  
    name : Name,  
    email : EmailAddress,  
    address : PostalAddress  
}
```

- What state is something in?

```
enum State {  
    NotConnected,  
    Connecting,  
    AuthenticationRequired,  
    LoggedIn,  
    ...  
}
```

```
struct AuthenticatedConnection {  
    socket : TcpSocket,  
    ...  
}
```

```
struct UnauthenticatedConnection {  
    socket : TcpSocket,  
    ...  
}
```

- Refine and extend the types as needed

Write the Functions (1/2)

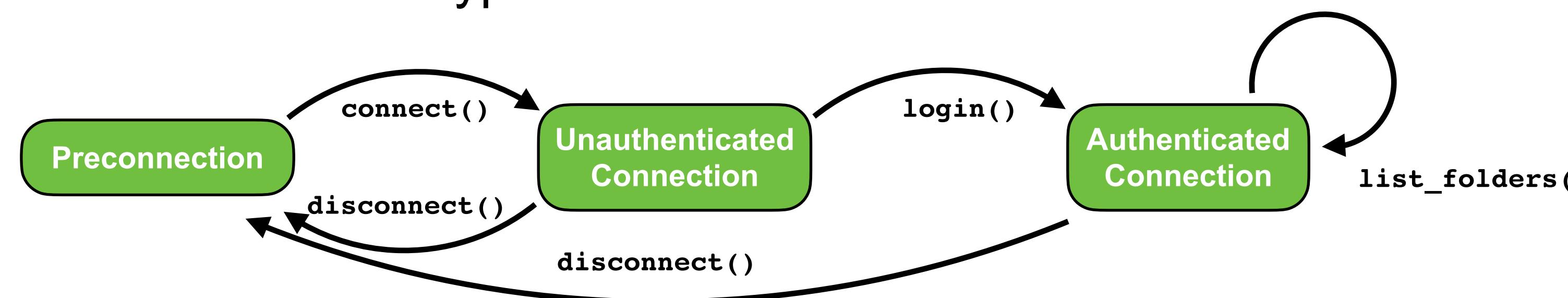
```
impl AuthenticatedConnection {  
    fn list_folders(self) -> List<EmailFolder> {  
        ...  
    }  
  
    fn list_messages(self, f : EmailFolder) -> List<EmailMessage> {  
        ...  
    }  
  
    fn disconnect(self) {  
        ...  
    }  
}
```

- Using the types as a guide, write the **function prototypes**, leaving the concrete implementation for later

```
impl UnauthenticatedConnection {  
    fn login(self, c: Credentials) -> Result<AuthenticatedConnection, LoginError> {  
        ...  
    }  
  
    fn disconnect(self) {  
        ...  
    }  
}
```

Write the Functions (2/2)

- Behaviour obvious from the types – and types constrain behaviour
 - Use specific rather than generic types
 - e.g., take a **Username** as a parameter, rather than a **String**
 - Types provide machine checkable documentation
 - Encode states as types and state transitions as functions



- State transitions return new types: **login()** consumes an **UnauthenticatedConnection**, returns an **AuthenticatedConnection** object
- Functions only implemented for the types where they make sense
 - e.g., **list_folders()** only implemented on **UnauthenticatedConnection**, so can't be called until after successful **login()**

Refine Types and Functions

- **Types and functions** provide a model of the system
- Iterate – filling in just enough details to keep it compiling
 - **Interactive design** using the compiler to check consistency
 - Gradually refine until the entire system is modelled – then add the concrete implementations, refining as needed
 - Work with the compiler to validate the design, before detailed implementation

Correct by Construction

- Use types to check the design, debugging before you run the code
- Non-sensical operations don't cause a crash – they don't compile
- Change of perspective: the compiler is a model checking tool that can help validate your design

Type-driven Development

- Define the types
- Write the functions
- Refine as needed

Design Patterns

- Specific numeric types
- Enumerations for alternatives

Numeric Types

- Is a value really a **double** or **int**, or does it have some meaning?
 - Temperature in degrees celsius
 - Speed in miles per hour
 - UserID
 - Packet sequence number
 - ...
- Encode the meaning as a type, so the compiler checks for consistent usage
 - Operations that mix types should fail, or perform safe unit conversions if possible
 - Operations that are inappropriate for a type shouldn't be possible

The screenshot shows the BBC Online Network homepage with the BBC News banner. The main headline reads "Sci/Tech Confusion leads to Mars failure". Below the headline is a photograph of a satellite, identified as the Mars Climate Orbiter, lying on its side on the reddish surface of Mars. A caption below the photo states: "The Mars Climate Orbiter: Now in pieces on the planet's surface. The Mars Climate Orbiter Spacecraft was lost because one Nasa team used imperial units while another used metric units for a key spacecraft operation." To the left of the main content area is a sidebar with navigation links for various BBC news sections like World, UK, Politics, Business, Sci/Tech, etc., and a "Feedback" link at the bottom.

Numeric Types – Strong Typing

- Weakly typed – programmer knows **c** and **f** are different types, but the compiler does not
- Program silently calculates the wrong answer

```
fn main() {  
    let c = 15.0; // Celsius  
    let f = 50.0; // Fahrenheit  
  
    let t = c + f;  
  
    println!("{}:{}", t); // 65.0  
}
```

Numeric Types – Strong Typing

- Strongly typed – **struct** with single unnamed field wraps numeric value
 - **derive** and **impl** standard operations
https://crates.io/crates/newtype_derive has macros to auto-generate **impl** blocks
 - Resulting code won't compile → types are mismatched

```
> cargo build
   Compiling foo v0.1.0 (/Users/csp/tmp/foo)
error[E0308]: mismatched types
--> src/main.rs:28:17
28   let t = c + f;
          ^ expected struct `Celsius`, found struct `Fahrenheit`
= note: expected type `Celsius`
         found type `Fahrenheit`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
error: Could not compile `foo`.

To learn more, run the command again with --verbose.
>
```

```
use std::ops::Add;

#[derive(Debug, PartialEq, PartialOrd)]
struct Celsius(f32);

#[derive(Debug, PartialEq, PartialOrd)]
struct Fahrenheit(f32);

impl Add for Celsius {
    type Output = Celsius;

    fn add(self, other : Celsius) -> Self::Output {
        Celsius(self.0 + other.0)
    }
}

impl Add for Fahrenheit {
    type Output = Fahrenheit;

    fn add(self, other : Fahrenheit) -> Self::Output {
        Fahrenheit(self.0 + other.0)
    }
}

fn main() {
    let c = Celsius(15.0);
    let f = Fahrenheit(50.0);
    let t = c + f;

    println!("{}:{}", t);
}
```

Numeric Types – Conversion

- We can add implementations that perform unit conversion

```
impl Add<Fahrenheit> for Celsius {
    type Output = Celsius;

    fn add(self, other: Fahrenheit) -> Self::Output {
        Celsius(self.0 + ((other.0 - 32.0) * 5.0 / 9.0))
    }
}

fn main() {
    let c = Celsius(15.0);
    let f = Fahrenheit(50.0);

    let t = c + f;

    println!("{}: {}", t); // Celsius(42.7) = Fahrenheit(109)
}
```

Numeric Types – Operations

- Do all the standard operations make sense for the type?
 - It's reasonable to compare **Celsius** values:

```
fn is_freezing(temp: Celsius) -> bool {  
    temp < Celsius(0.0)  
}
```

so you'd implement the **Ord** trait that provides these operations

- But this might not make sense for a **UserID** type
 - You likely want to be able to compare two **UserID** values for equality (the **Eq** trait), but adding two **UserID** values or comparing to see which is largest might not be meaningful
 - Not all standard operations need to be implemented for a type

Numeric Types – No Runtime Cost

- Wrapping value inside **struct** adds zero runtime overhead in Rust
 - Programmer must implement standard operations – some extra code, but no runtime cost
 - https://crates.io/crates/newtype_derive provides macros to implement the common cases
 - Why no runtime cost?
 - No information added to the **struct**, so same size
 - Passed in the same way – not automatically boxed on the heap
 - Optimiser will recognise that the code collapses down to operations on primitive types, and generate the code to do so
 - All the additions are a compile-time model of the ways the data can be used, they don't affect the compiled code
 - (*Equivalent C++ code has the same properties*)

Alternative Types: enum

- Use `enum` types and pattern matching to model alternatives and options
 - `Option<T>`
 - `Result<T,E>`
 - Features and response codes
 - Flags
- The compiler can check that options, alternatives, features, and flags are handled correctly – debug before running the code

Optional Values: Option<T>

- If a value might not exist, use **Option<T>**
 - As function return value, pattern matching on result:

```
fn lookup(self, user : Username) -> Option<User>
```

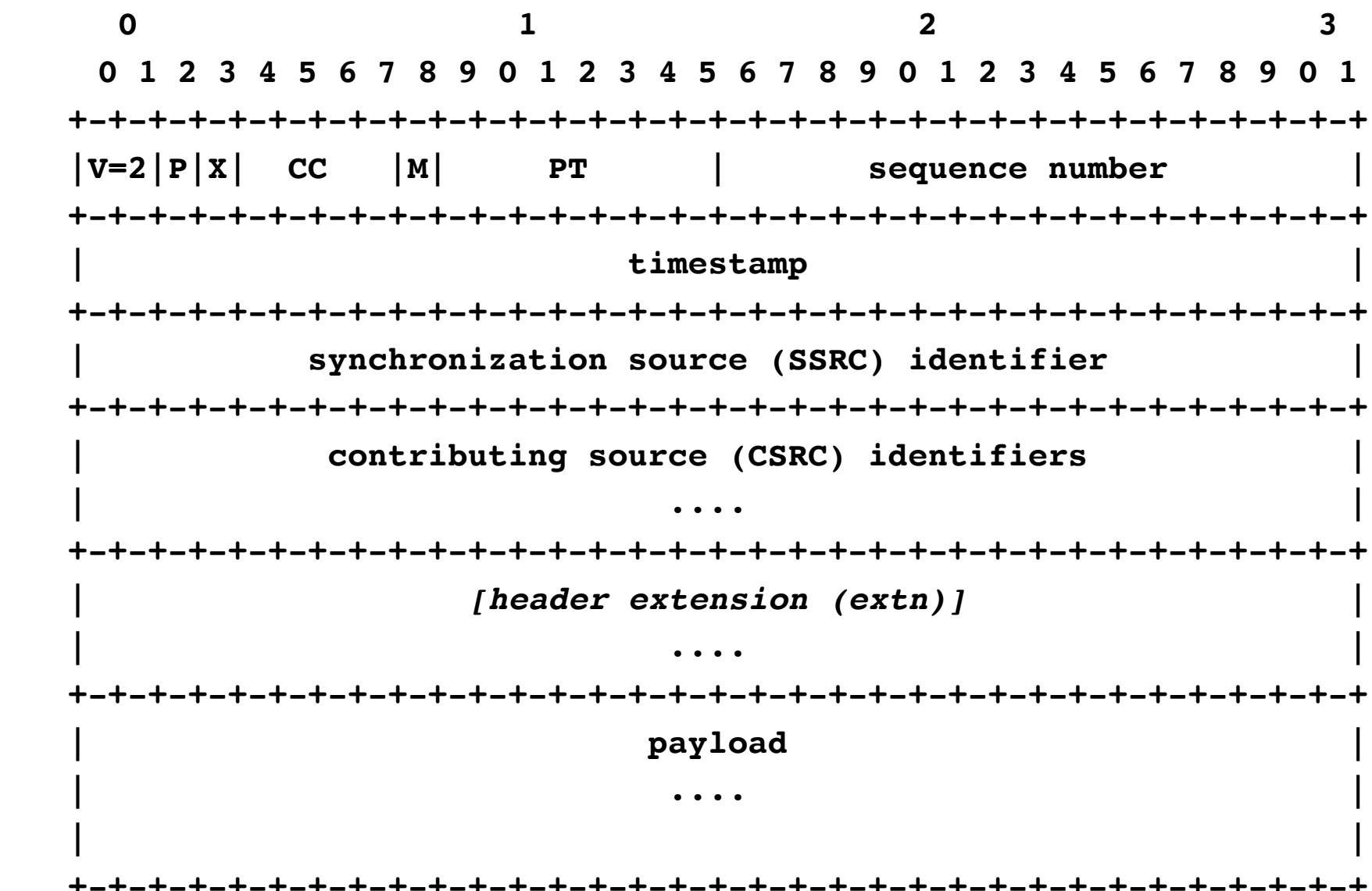
Optional Values: Option<T>

- If a value might not exist, use **Option<T>**
 - As function return value, pattern matching on result:

```
fn lookup(self, user : Username) -> Option<User>
```

- In **struct** definition:

```
struct RtpHeader {  
    v      : Version,  
    pt     : PayloadType,  
    seq    : SequenceNumber,  
    ts     : Timestamp,  
    ssrc   : SourceId,  
    csrc   : Vec<SourceId>,  
    extn   : Option<HeaderExtension>,  
    payload : RtpPayload  
}
```



Optional Values: Option<T>

- If a value might not exist, use **Option<T>**
 - As function return value, pattern matching on result:

```
fn lookup(self, user : Username) -> Option<User>
```

- In **struct** definition:

```
struct RtpHeader {  
    v      : Version,  
    pt     : PayloadType,  
    seq    : SequenceNumber,  
    ts     : Timestamp,  
    ssrc   : SourceId,  
    csrc   : Vec<SourceId>,  
    extn   : Option<HeaderExtension>,  
    payload : RtpPayload  
}
```

- Compiler enforces that both variants of **Option<T>** are handled
 - **Some(T)**, **None**
 - Can't accidentally write code that assumes the value is present and crashes otherwise

Results: **Result<T, E>**

- The **Result<T, E>** type represents results that can fail
- Used as a result type for a function:

```
fn load_document() -> Result<Document, DatabaseError> {
    let db = open_database()?;
    db.load("document_1")?
}
...
match load_document() {
    Ok(doc) => println!(doc), // success
    Err(e)   => ...           // failed
}
```

Use of ? operator for early return on error

Custom error types can be defined: https://doc.rust-lang.org/rust-by-example/error/multiple_error_types/define_error_type.html

- C code frequently uses a signal value to indicate errors
 - e.g., **socket()** returns -1 on error, file descriptor >0 on success
 - Success and failure have the same type – easy to forget to check error codes
- With **Result<T, E>**, must pattern match to distinguish success and failure type; compiler will check that all error codes are handled

Features and Response Codes

- Anti-pattern #1: string typing
 - Method parameters that are strings, rather than some more appropriate type
 - Strings returned from network function (e.g., HTTP response codes) directly used, rather than converted to appropriate type
- Use `enum` to represent values that can be one of several alternatives
 - Exhaustiveness checking – catches bugs if new codes introduced
 - Ease of refactoring – decouples code from external representation
 - Make nonsensical values unrepresentable
 - Types are machine checkable documentation – strings are not

Flags

- Anti-pattern #2: boolean flags
 - Use of boolean flags obscures code meaning – compare:

```
let f = File.Open("foo.txt", true, false);
```

and

```
let f = File.Open("foo.txt", FileMode::TextMode, FileMode::ReadOnly);
```

- Further, the **enum**-based version won't compile if the arguments are passed in the wrong order; the **bool**-based version will compile, run, and give the wrong results
- See also:
 - <https://www.luu.io/posts/dont-use-booleans/>
 - <https://wiki.c2.com/?UseEnumsNotBooleans>

Design Patterns

- Use the type system to describe features of the system design, so the compiler can check for correctness
- There is an up-front cost: you must define the types
- The benefit is that fixing compilation errors is easier than fixing silent data corruption
 - For small systems, the cost may outweigh the benefit
 - For large systems, compiler enforced consistency checks due to use of types can be a significant win

Design Patterns

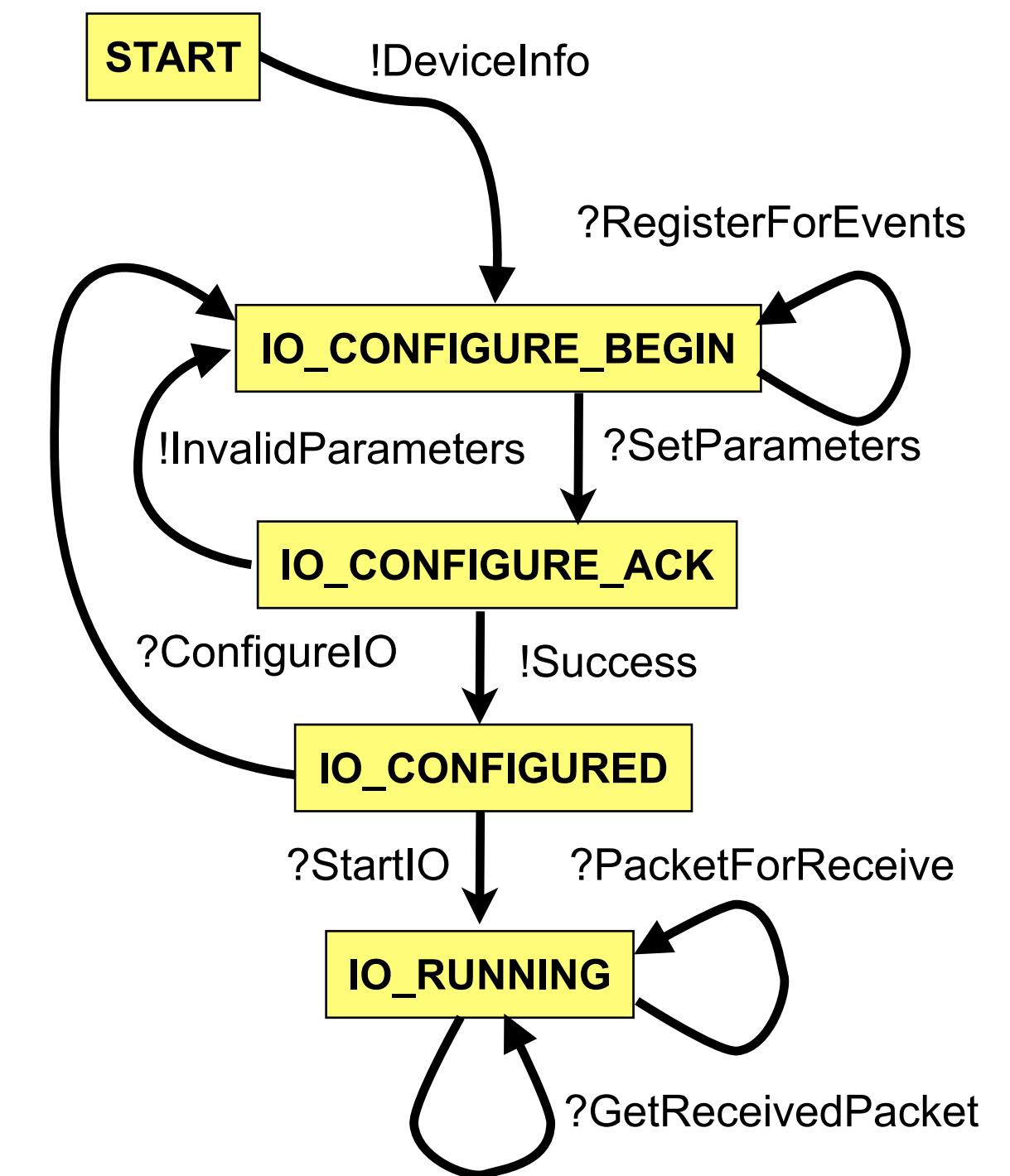
- Specific numeric types
- Enumerations for alternatives

State Machines

- What is a state machine?
- Implementation using **enum** types
- Implementation using **struct** types

State Machines

- State machines common in systems code
 - Network protocols
 - File systems
 - Device drivers
- System behaviour modelled as a finite state machine comprising:
 - **States** that reflect the status of the system
 - **Events** that trigger transitions between states
 - **State variables** that hold system configuration
- Clean high-level model of a system
 - Captures the essence of the behaviour
 - Easy to reason about and prove properties such as termination, absence of deadlocks, reachability, etc.



Implementing State Machines

- Hard to cleanly model state machine in code
 - Structure of code tends not to match structure of state machine; not easy to visualise transitions
 - Difficult to validate code against specification
- Approaches to modelling state machines in strongly-typed functional languages:
 - Encode states and events as enumerations, pattern match on (state, event) tuples
 - Encode states as types and transitions as functions
 - Add first-class state machine support to language
 - Microsoft Singularity research operating system
 - **async/await** asynchronous code → lecture 8

```
contract NicDevice {  
    out message DeviceInfo(...);  
    in  message RegisterForEvents(NicEvents.Exp:READY  
c);  
    in  message SetParameters(...);  
    out message InvalidParameters(...);  
    out message Success();  
    in  message StartIO();  
    in  message ConfigureIO();  
    in  message PacketForReceive(byte[] in ExHeap p);  
    out message BadPacketsize(byte[] in ExHeap p, int  
m);  
    in  message GetReceivedPacket();  
    out message ReceivedPacket(Packet * in ExHeap p);  
    out message NoPacket();  
  
    state START: one {  
        DeviceInfo! → IO_CONFIGURE_BEGIN;  
    }  
    state IO_CONFIGURE_BEGIN: one {  
        RegisterForEvents? →  
        SetParameters? → IO_CONFIGURE_ACK;  
    }  
    state IO_CONFIGURE_ACK: one {  
        InvalidParameters! → IO_CONFIGURE_BEGIN;  
        Success! → IO_CONFIGURED;  
    }  
    state IO_CONFIGURED: one {  
        StartIO? → IO_RUNNING;  
        ConfigureIO? → IO_CONFIGURE_BEGIN;  
    }  
    state IO_RUNNING: one {  
        PacketForReceive? → (Success! or BadPacketsize!)  
        → IO_RUNNING;  
        GetReceivedPacket? → (ReceivedPacket! or  
        NoPacket!)  
        → IO_RUNNING;  
        ...  
    }  
}
```

Listing 1. Contract to access a network device driver.

G. Hunt and J. Larus. “Singularity: Rethinking the software stack”, ACM SIGOPS OS Review, 41(2), April 2007. DOI:10.1145/1243418.1243424

Enumerations for modelling state machines

- Possible state machine representation:
 - An enumerated type (**enum**) models alternatives
 - Define an **enum** to represent the states
 - Define an **enum** to represent the events
 - Functions represent transitions and actions:
 - Define a function to map from (state, events) tuples to next state
 - Define a function to perform the actions associated with each state
 - Builds on the intuition that **enum** types express alternatives, and a state machine comprises a list of alternative states

Using enum to Model State Machines: Example (1/3)

```
enum ApcState {  
    Initialize,  
    WaitForConnect,  
    Accept(TcpStream),  
    StartTransfer(TcpStream),  
    Waiting(TcpStream),  
    ReceiveMsg(TcpStream, Vec<u8>),  
    SendNop(TcpStream),  
    Closed,  
    Finish,  
    Failure(String),  
}
```

```
enum ApcEvent {  
    TcpConnected(TcpStream),  
    ResponseValid(bool),  
    IncomingTcpClosed,  
    AspMsgIn(Vec<u8>),  
    NopTimeout,  
    Finished,  
    Uct,  
}
```

Example adapted from comment on
<https://hoverbear.org/2016/10/12/rust-state-machine-pattern/>

- Define an **enum** to represent states and another for the events
- Encode state variables as **enum** parameters

Using enum to Model State Machines: Example (2/3)

```
impl ApcState {
    fn next(self, event: ApcEvent) -> Self {
        use self::ApcState::*;
        use self::ApcEvent::*;

        match (self, event) {
            (Initialize, TcpConnected(tcp)) => Accept(tcp),
            (Initialize, Finished) => Finish,
            (Accept(tcp), ResponseValid(true)) => StartTransfer(tcp),
            (Accept(_), ResponseValid(false)) => Closed,
            (StartTransfer(tcp), Uct) => Waiting(tcp),
            (Waiting(_), IncomingTcpClosed) => Closed,
            (Waiting(_), Finished) => Finish,
            (Waiting(tcp), AspMsgIn(msg)) => ReceiveMsg(tcp, msg),
            (Waiting(tcp), NopTimeout) => SendNop(tcp),
            (ReceiveMsg(tcp, _), Uct) => Waiting(tcp),
            (SendNop(tcp), Uct) => Waiting(tcp),
            (s, e) => Failure(format!("Invalid State/Event combination: {:#?}/{:#?}", s, e)),
        }
    }
}
```

- Match against states and events: clean representation of state-transition table
- Straight-forward to validate against specification

Using enum to Model State Machines: Example (3/3)

```
pub struct ApcStateMachine {
    state : ApcState,
    addr : SocketAddr,
    timeout: u64,
}

impl ApcStateMachine {
    fn new() -> ApcStateMachine {
        ...
    }

    fn run_once(&self) -> ApcEvent {
        match self.state {
            Initialize      => ...
            WaitForConnect  => ...
            Accept(tcp)     => ...
            StartTransfer(_) => ...
            Waiting(tcp)   => ...
            ReceiveMsg(_, msg) => ...
            SendNop(_)      => ...
            closed          => ...
            Finish          => ...
        }
    }
}
```

```
fn run_state_machine() {
    let mut sm = ApcStateMachine::new();
    loop {
        let event = sm.run_once();
        sm.state = sm.state.next(event);
        if sm.state == ApcState::Finish {
            break;
        }
    }
}
```

- **match** loop dispatches to functions
 - Performs actions each state, returns next event to process → determine next state
 - Parameterised enum with state variables makes it easy to pass parameters
- Pattern matching on **enum** gives a clear implementation
 - Compiler checks all alternates covered
 - Easy to pass state variables

Structures for Modelling State Machines

- Alternative state machine representation:
 - Define a **struct** representing each state
 - Model an event as a method call on a **struct**
 - Model state transitions by returning a **struct** representing the new state
- Builds on the intuition that states hold concrete *state*, and events are things that happen in states

Using `struct` to Model State Machines: Example

- Define a `struct` representing each state
 - State variables are fields within the `struct`
 - Methods implemented on the `struct` encode state transitions and event handlers
 - Return `Self` if state is unchanged
 - Return `struct` representing new state if state changes

```
impl UnauthenticatedConnection {
    fn login(self, c: Credentials) -> Result<AuthenticatedConnection, (self, LoginError)> {
        ...
    }

    fn disconnect(self) -> NotConnected {
        ...
    }
}
```

```
struct UnauthenticatedConnection {
    socket : TcpSocket,
    ...
}

struct AuthenticatedConnection {
    socket : TcpSocket,
    ...
}

struct NotConnected;
```

- Encodes states and state transitions in types and *ownership rules*

Approaches to Representing State Machines

- **enum**-based approach is compact, makes states and events clear in the types, and has clear state transition table
 - Relies on expressive **enum** types for implementation – works well in languages like Rust, Swift, OCaml, ...
 - Difficult to express in languages with weaker **enum** types and pattern matching
- **struct**-based approach encodes states and state transitions in the types, events as methods on those types
 - State transition table is less obviously explicit in the code
 - State transitions use Rust ownership rules to enforce transitions – cannot easily check state transitions in languages without explicit data ownership

State Machines

- What is a state machine?
- Implementation using **enum** types
- Implementation using **struct** types

Ownership

- Ownership of data in Rust
- Enforcing state transitions

Ownership

- Systems programs care about ownership of resources
 - To control memory management, close files, etc. → lecture 5
 - To model state machines
- Programmer maintains a mental model of what part of the code owns each resource
 - What function is responsible for calling `free()`, `close()`, etc.
 - Garbage collected languages still require understanding of ownership – but make `free()` call automatic when lifetime ends
 - C++ and Python tie resource ownership to scoping:

```
with open(filename) as file:  
    data = file.read()  
    ...
```

gives automatic resource clean-up at end of scope

Ownership in Rust

- Rust tracks ownership of data – enforces that every value has a single owner
- Function calls explicitly manage ownership of values
 - Take explicit ownership of a value

```
fn consume(r : Resource) {  
    ...  
}
```

Function *takes ownership* of parameter passed by value
No longer accessible to caller; freed at end of function

- Borrow a value

```
fn borrow(r : &Resource) {  
    ...  
}
```

Function *borrow*s the parameter passed via reference
Ownership remains with caller

- Return ownership of a value

```
fn generate() -> Resource {  
    ...  
}
```

Function *passes ownership* of return value to caller

Ownership in Rust: Example

```
struct Resource {  
    value : u32  
}
```

```
fn consume(r : Resource) {  
    println!("consumed");  
}  
  
fn main() {  
    let r = Resource{value: 42};  
    consume(r);  
    println!("{}", r.value);  
}
```

Function *takes ownership* of parameter passed by value
No longer accessible to caller; freed at end of function

- The **consume()** function takes ownership of the resource – doesn't return it to the caller
- Above code won't compile: **println!()** cannot access **r.value**, since **main()** no longer has access to **r** because it gave ownership to **consume()**

State Machines and Ownership

- State machines manage resources
 - A network protocol manages connections, and the data sent over them
 - A device driver manages hardware resource
 - ...
- State transitions indicate when resources created/go out-of-scope

Ownership and state machines (1/2)

- **struct**-based approach to state machines uses ownership rules to enforce state transitions
 - Methods that change state take ownership of **self**, return new **struct**:

```
impl UnauthenticatedConnection {  
    fn login(self, c: Credentials) -> Result<AuthenticatedConnection, (self, LoginError)> {  
        ...  
    }  
  
    fn disconnect(self) -> NotConnected {  
        ...  
    }  
}
```

- e.g., the **login()** function consumes its **UnauthenticatedConnection** and returns a new **AuthenticatedConnection** on success
 - The compiler enforces this – the **UnauthenticatedConnection** is not accessible after this call; all resources it owned are reclaimed
 - Except any values the **login()** method explicitly copies to the **AuthenticatedConnection**

Ownership and state machines (2/2)

- **struct**-based approach to state machines uses ownership rules to enforce state transitions
 - Guarantees resource cleanup on state transition
 - Better for ensuring resources are cleaned-up after use
- **enum**-based approach to state machines makes the state transition diagram clearer, but relies on programmer discipline to clean-up
 - Better for ensuring complex state machines correctly reflected in code

Type-driven Development – Recap

- **Define the types first**
 - Define concrete numeric types, identifiers
 - Define enum types to represent alternatives
 - Indicate optional values, results, error types

Type-driven Development – Recap

- **Define the types first**
- Using the types as a guide, **write the functions**
 - Write the input and output types
 - Write the function, using the structure of the types as a guide
 - Make state machines explicit
 - Consider ownership of data

Type-driven Development – Recap

- **Define the types first**
- Using the types as a guide, **write the functions**
- **Refine** and edit types and functions as necessary
 - Use the compiler as a tool to help you debug your design

Type-driven Development – Recap

- **Define the types first**
- Using the types as a guide, **write the functions**
- **Refine** and edit types and functions as necessary
- Don't think of the types as checking the code, think of them as a plan, a model, for the solution – as machine checkable documentation – use the compiler as a debugging tool before running the code

Summary

- Type-drive development
- Design patterns
- State machines
- Ownership

Resource Ownership and Memory Management

Advanced Systems Programming (H/M)
Lecture 5

Outline

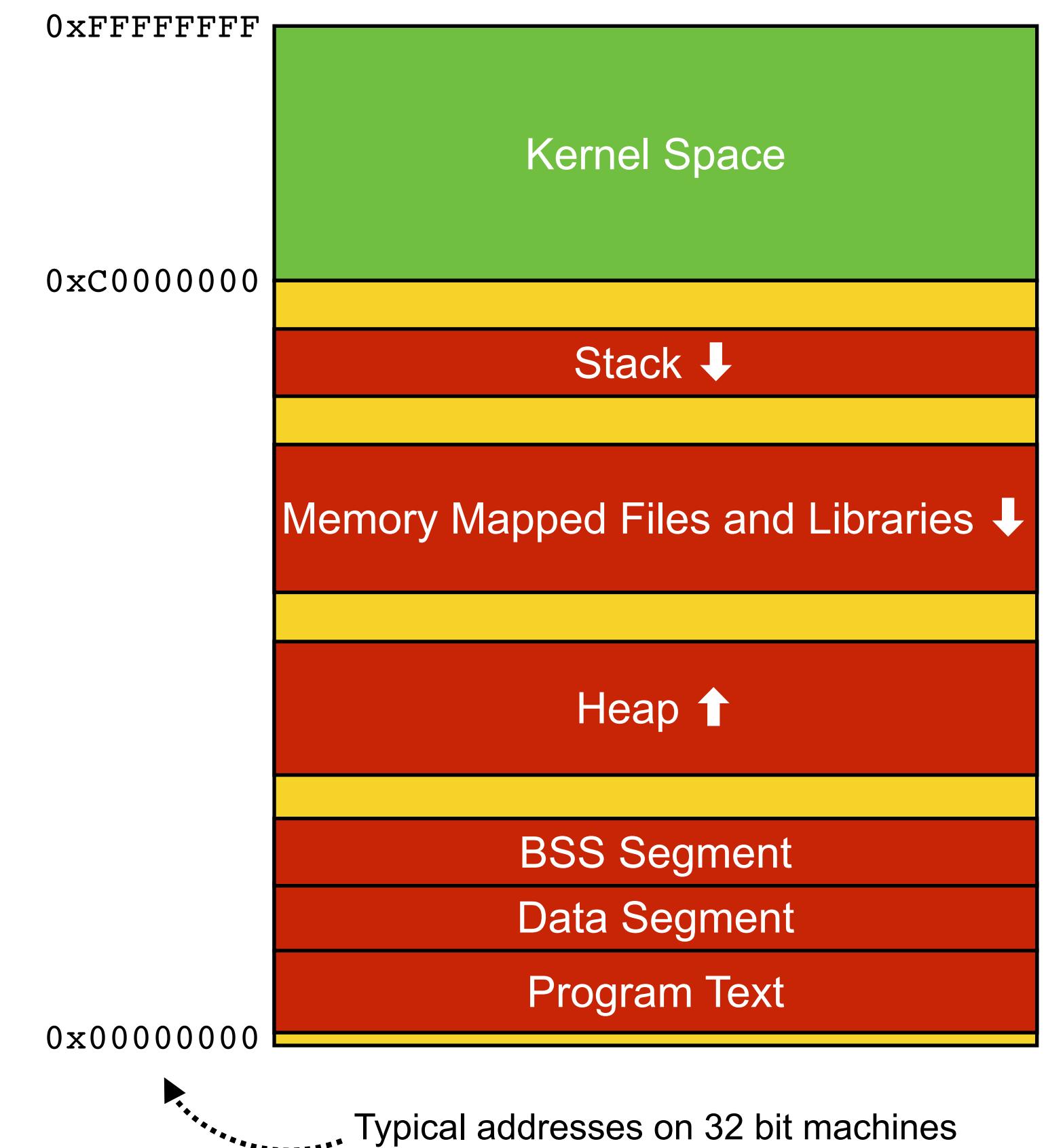
- Memory
 - How is a process stored in memory?
 - What memory has to be managed?
- Memory management
 - Reference counting
 - Region-based memory management
- Resource management

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

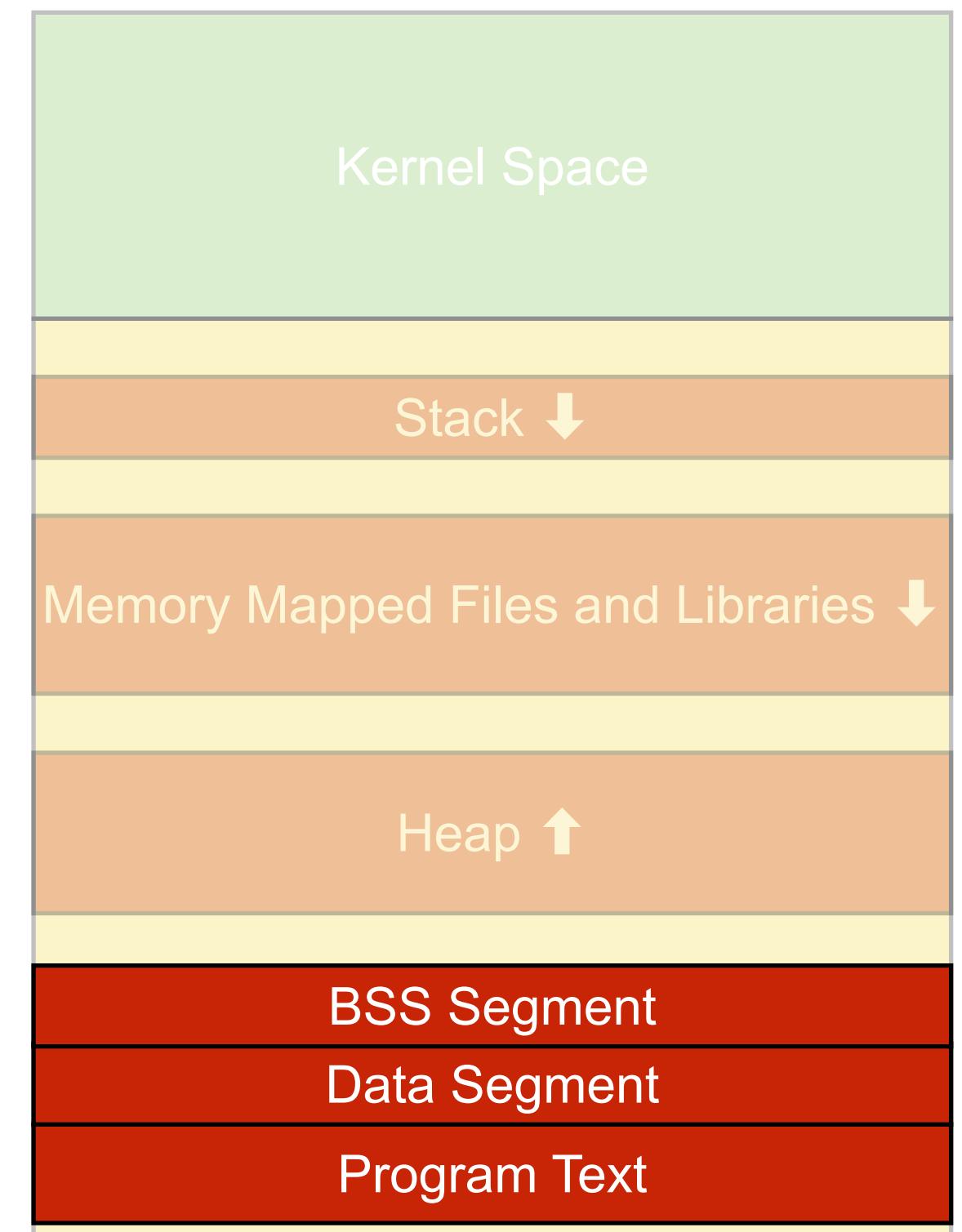


See also:

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

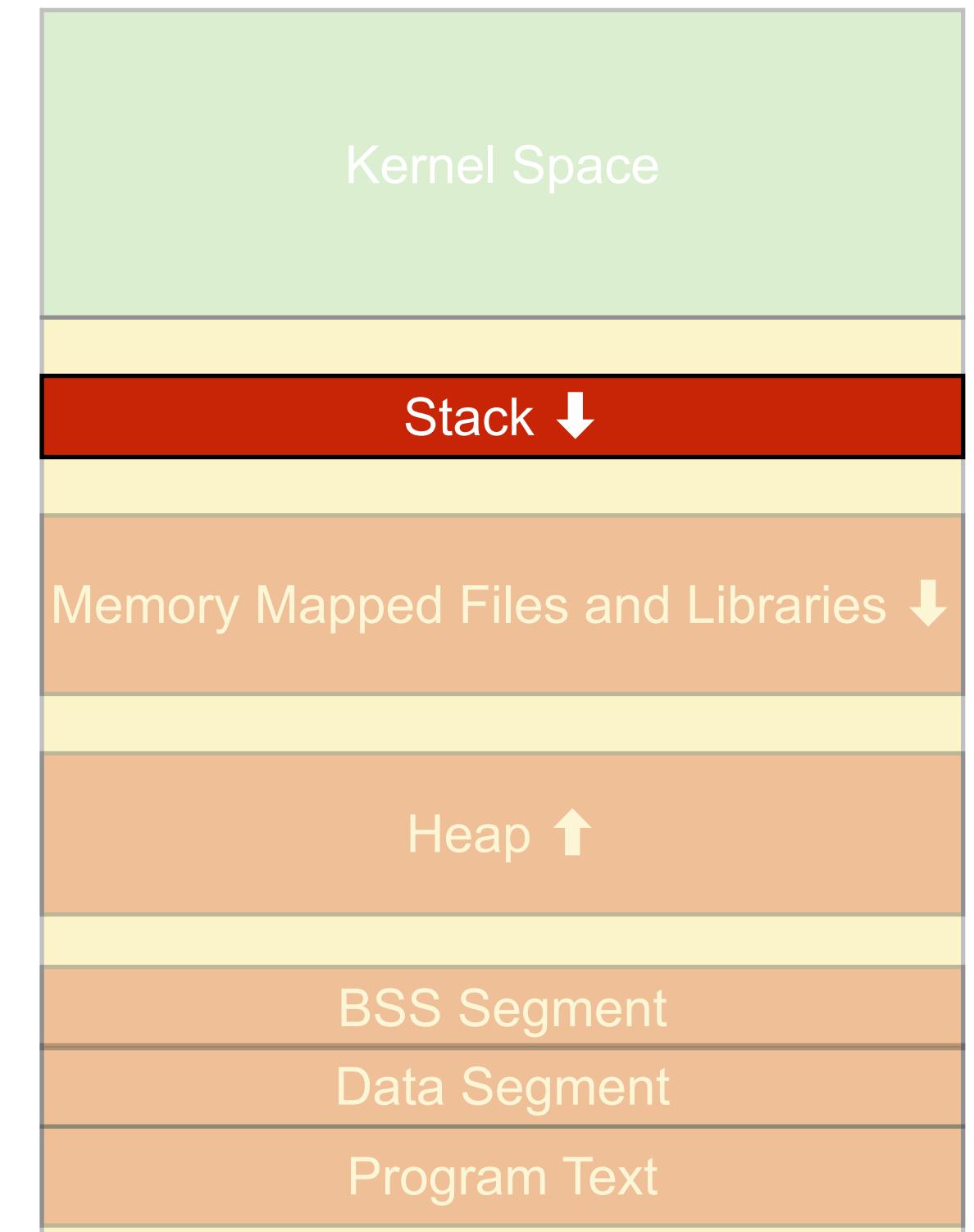
Program Text, Data, and BSS

- 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



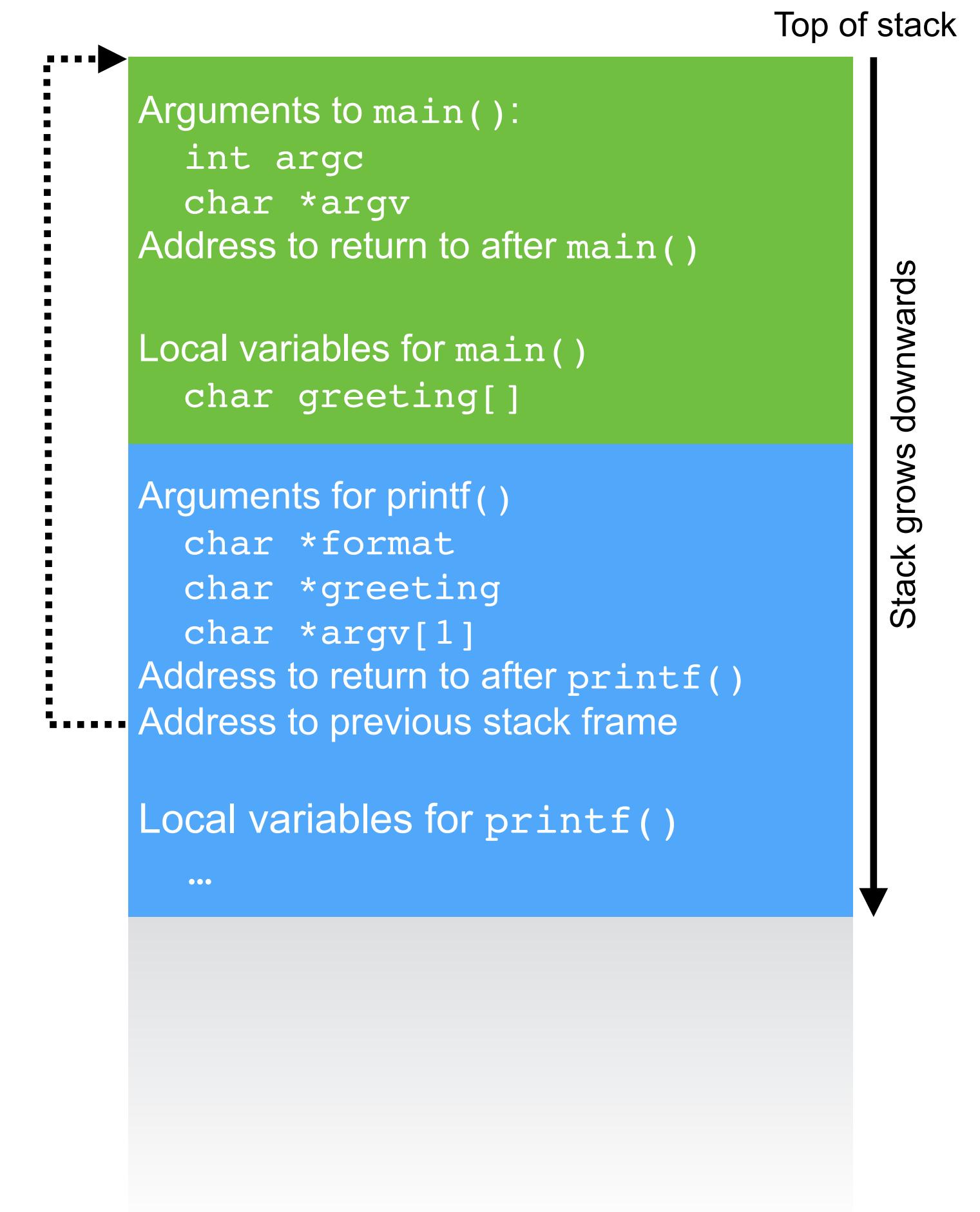
Function Calling Conventions

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

```
#include <stdio.h>

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

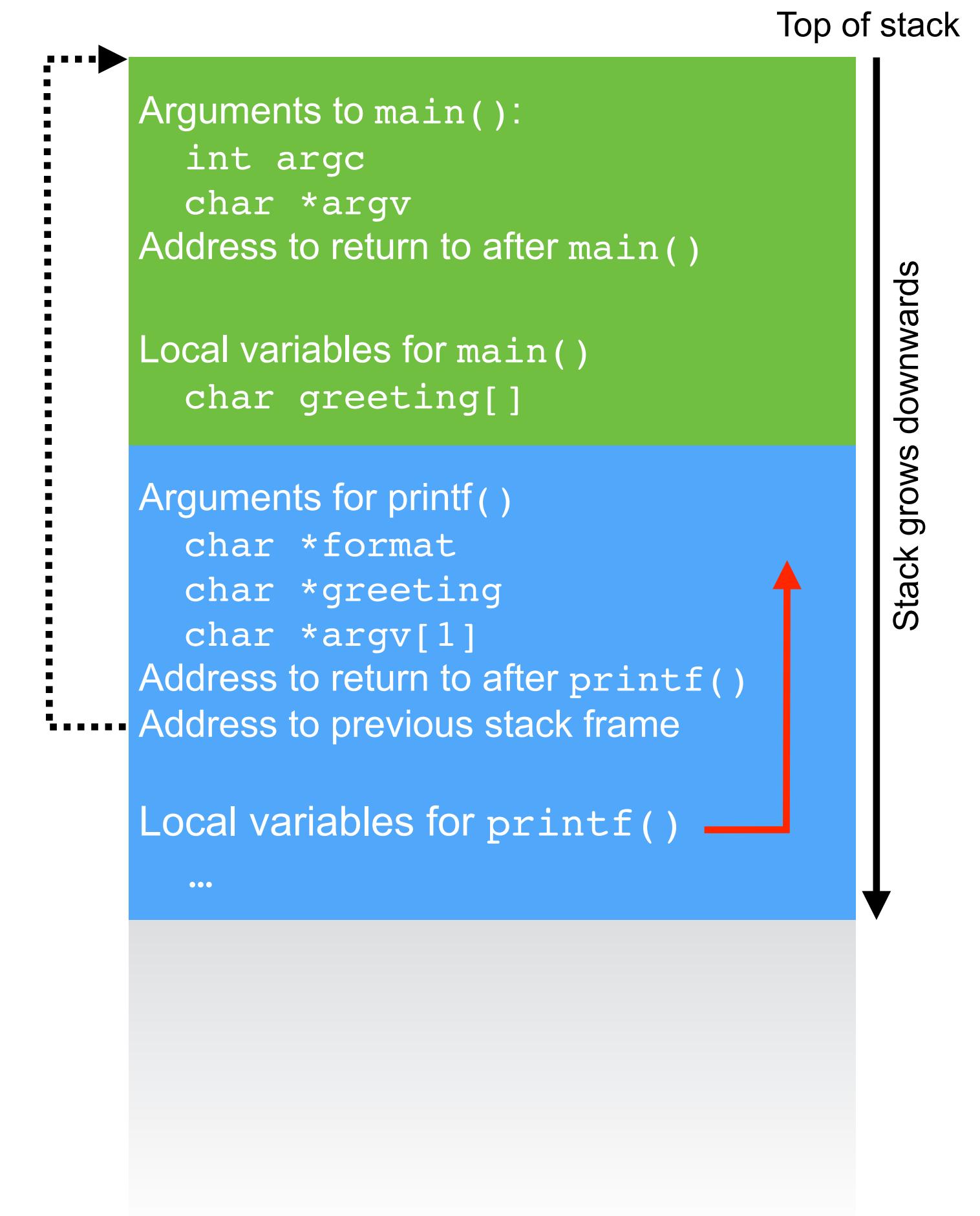
    if (argc == 2) {
        printf("%s, %s\n", greeting, argv[1]);
        return 0;
    } else {
        printf("usage: %s <name>\n", argv[0]);
        return 1;
    }
}
```



- 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

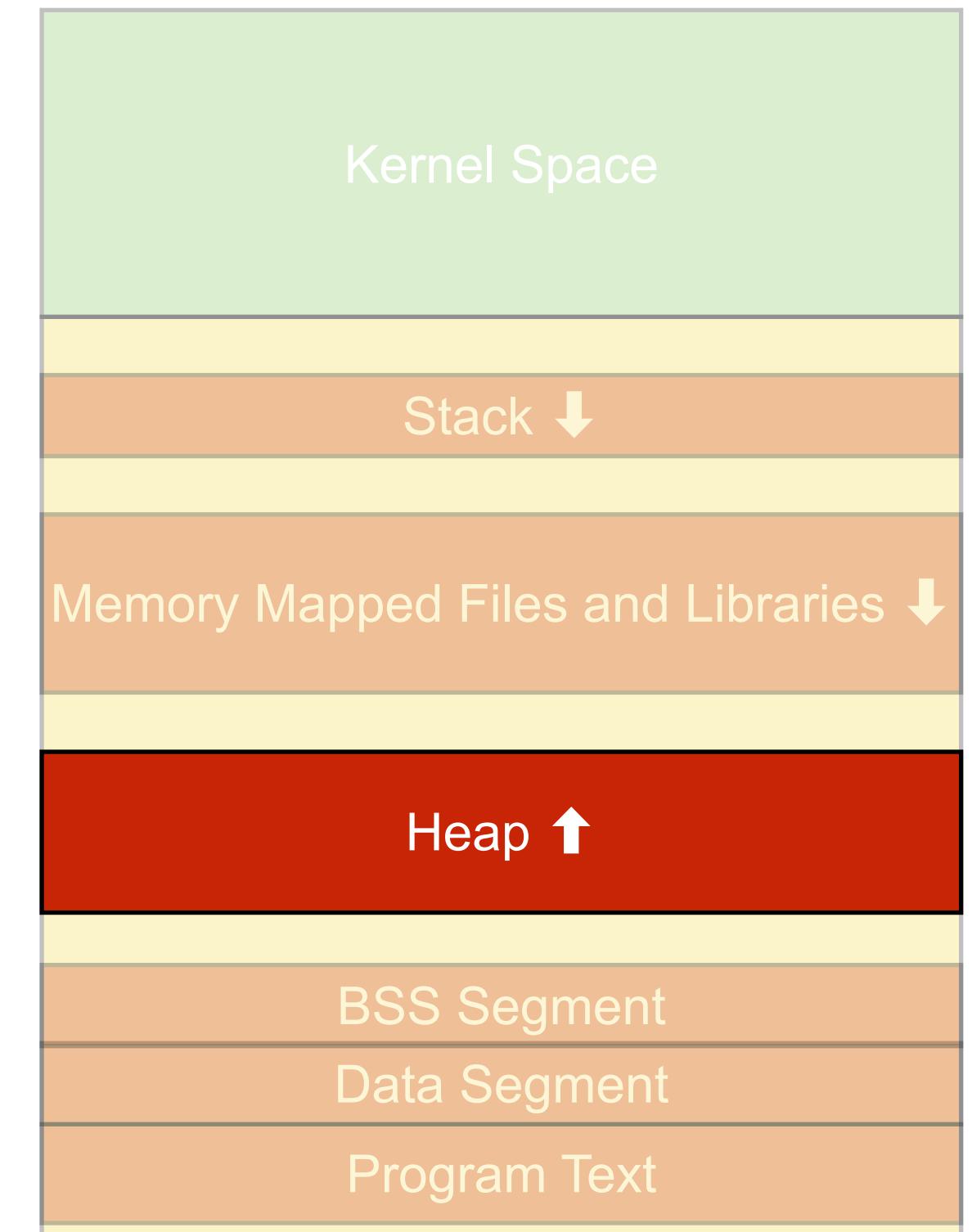
Buffer Overflow Attacks

- 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



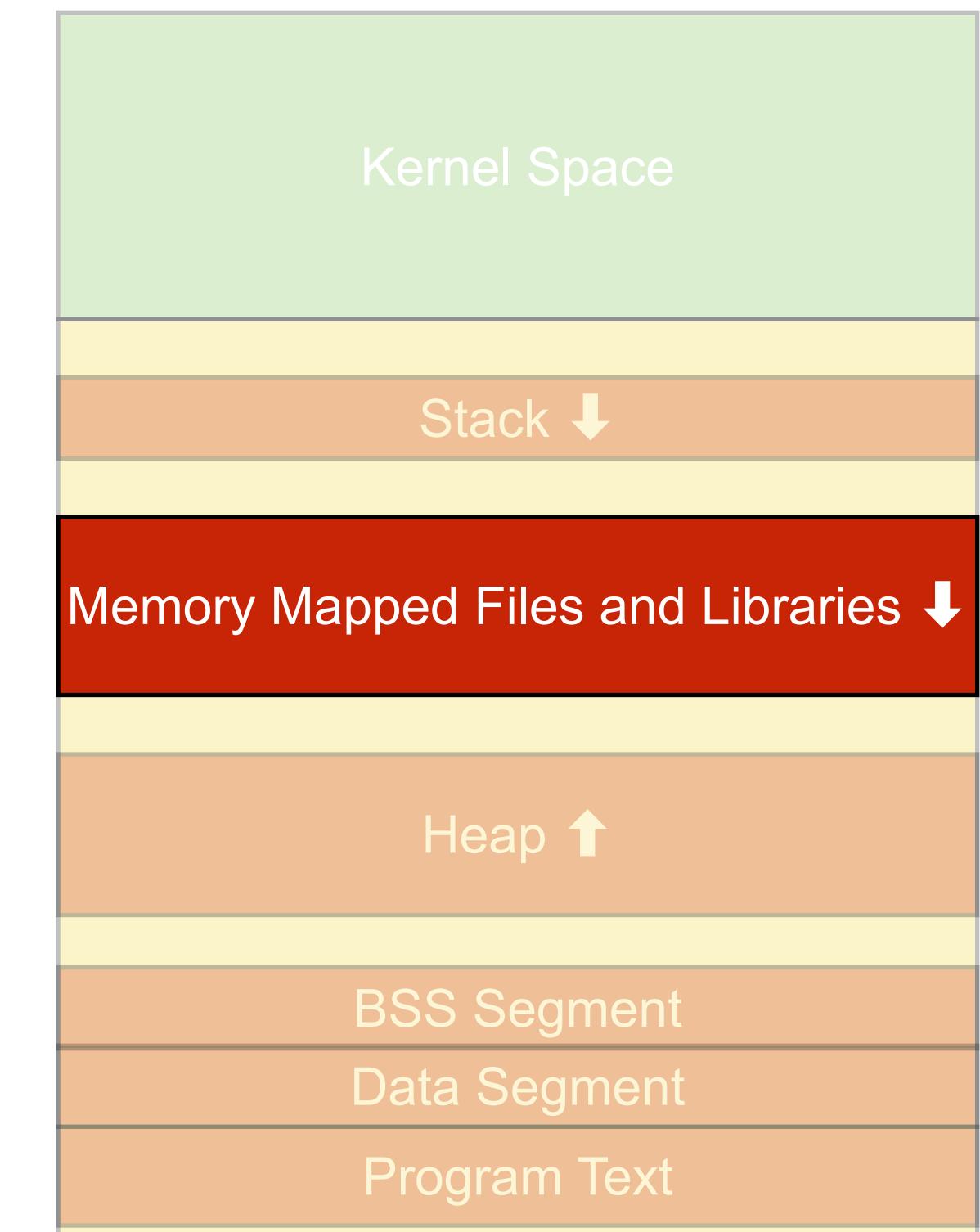
The Heap

- 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 for 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



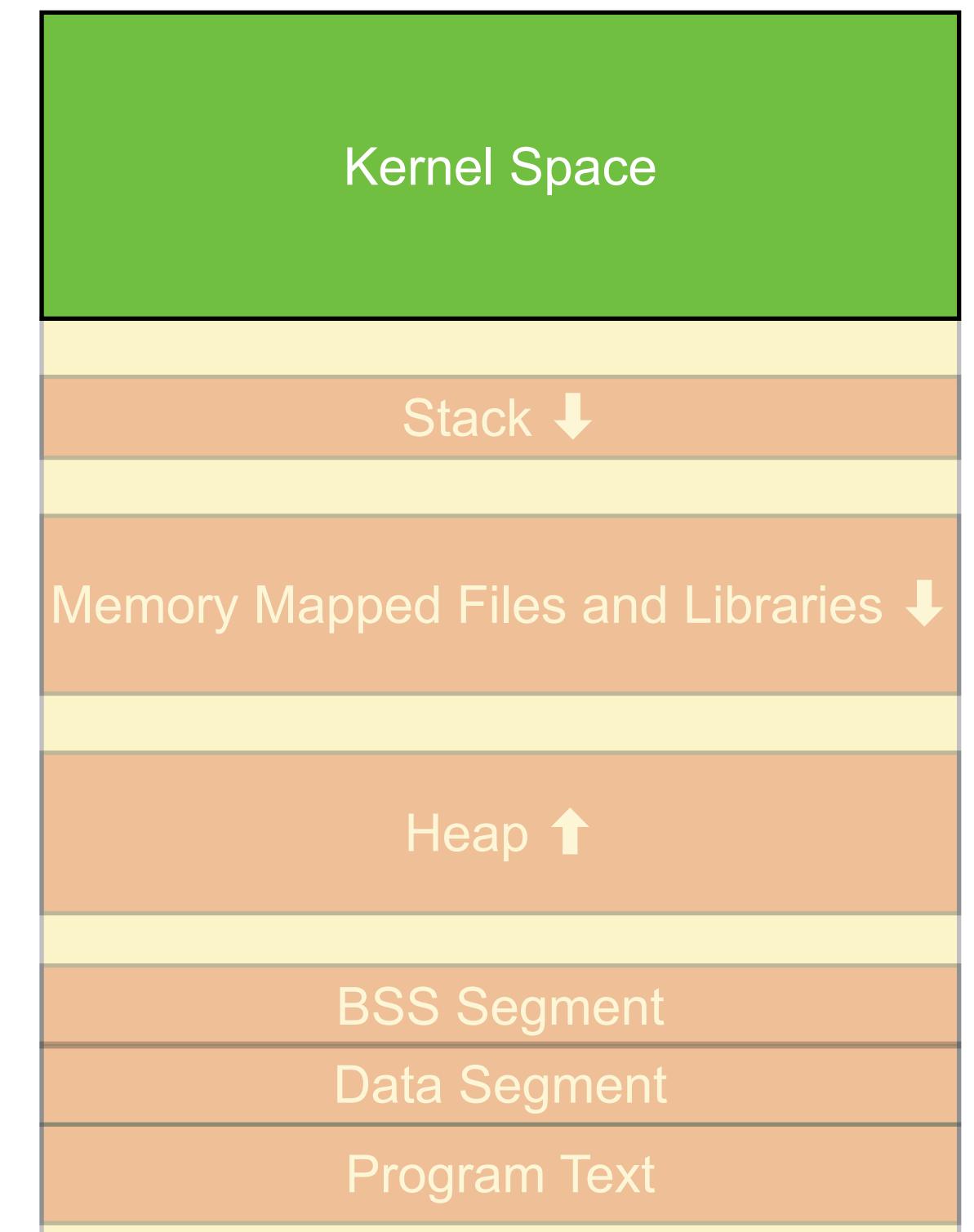
Memory Mapped Files and Shared Libraries

- **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?

Automatic Memory Management

- Concepts
- Approaches

Automatic Memory Management

- Automatic memory management distrusted by systems programmers
 - Perceived high processor and memory overheads, unpredictable timing
 - But, memory management problems are common:
 - Unpredictable performance
 - Calls to `malloc()`/`free()` can vary in execution time by several orders of magnitude
 - Memory leaks
 - Memory corruption and buffer overflows
 - Use-after-free
 - Iterator invalidation
- New automatic memory management schemes solve many problems
 - Garbage collectors → lower overhead, more predictable
 - Also system performance improvements made overhead more acceptable
 - Region-based memory management → predictability, compile time guarantees

Memory Management in Systems Programs

- Systems programs traditionally used a mix of manual and automatic memory management:
 - Stack memory managed automatically:
 - In the example, memory for `di` is automatically allocated when the function executes; freed on completion
 - Simple and efficient for languages like C/C++ that have complex value types
 - Less effective for Java-like languages, where objects always allocated on the heap
 - Heap memory managed manually:
 - Allocation using `malloc()`, deallocation using explicit `free()`

```
int saveDataForKey(char *key, FILE *outf)
{
    struct DataItem di;

    if (findData(&di, key)) {
        saveData(&di, outf);
        return 1;
    }
    return 0;
}
```

Managing the Heap

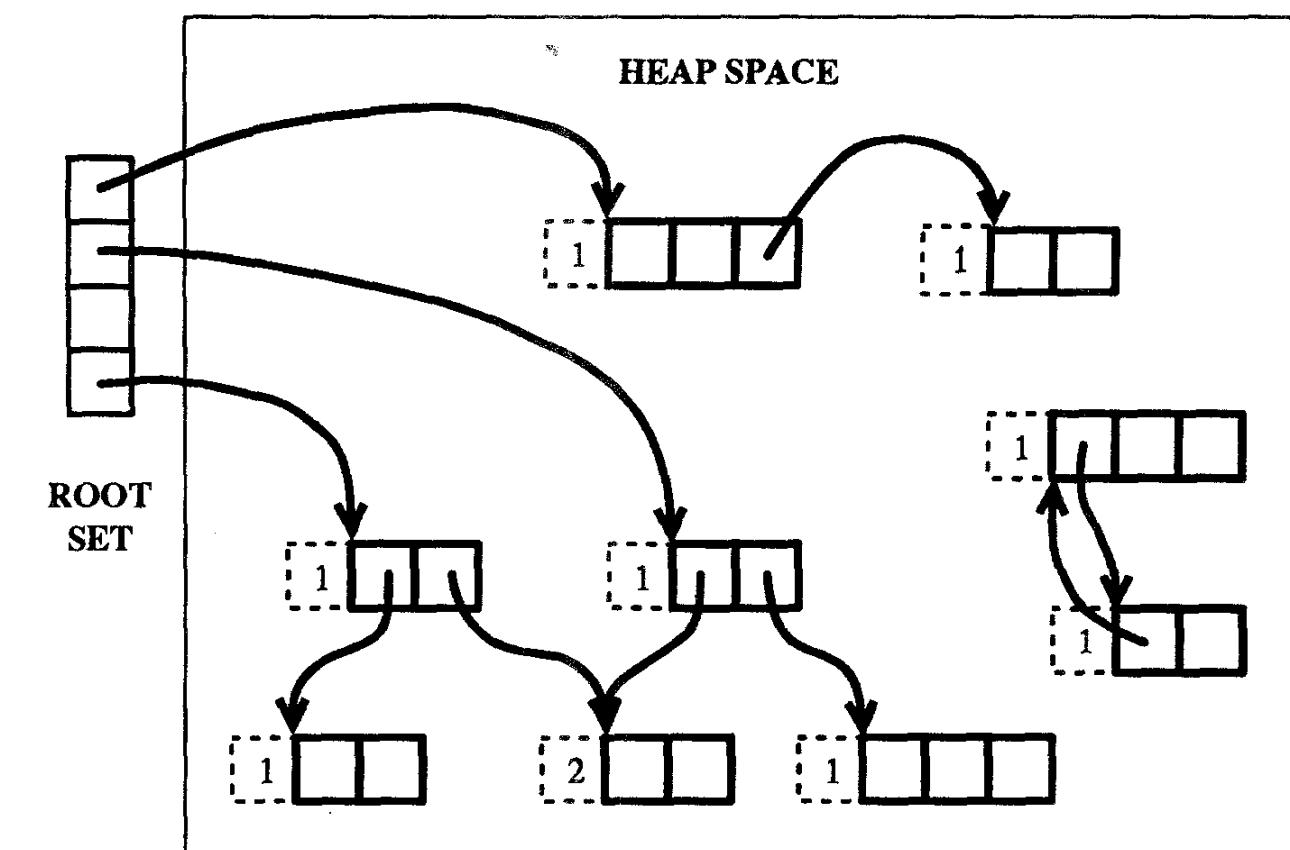
- Aim: to find objects that are no longer used, make their space available for reuse
 - An object is no longer used (ready for reclamation) if it is not reachable by the running program via any path of pointer traversals
 - Any object that is *potentially* reachable is preserved – better to waste memory than deallocate an object that's in use
- Approaches to automatic heap management:
 - Reference counting
 - Region-based lifetime tracking
 - Garbage collection → lecture 6

Reference Counting

- Reference counting
- Costs and benefits

Reference Counting

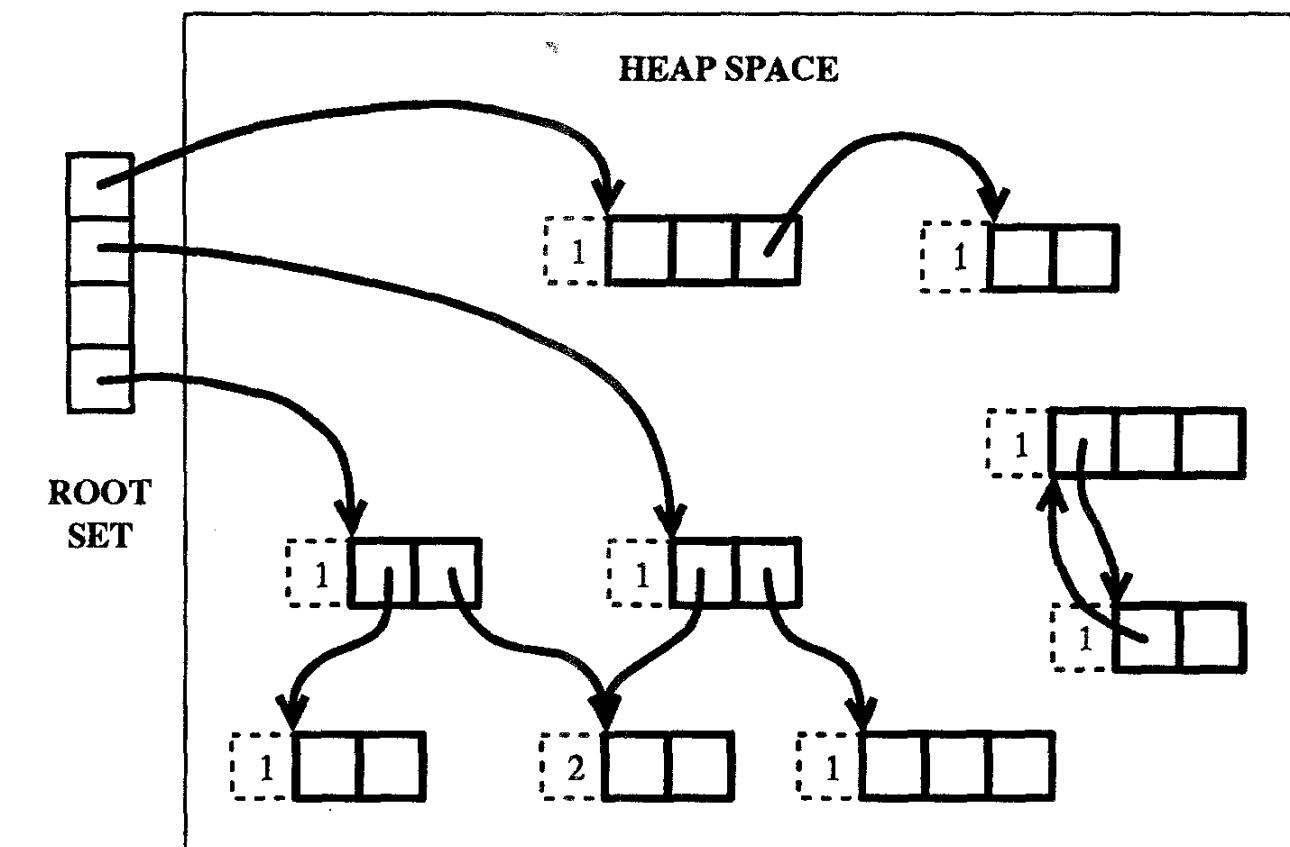
- Simplest automatic heap management
- Allocations contain space for an additional **reference count**
 - An extra `int` is allocated along with every object
 - Counts number of references to the object
 - Increased when new reference to the object is created
 - Decremented when a reference is removed
 - When reference count reaches zero, there are no references to the object, and it may be reclaimed
 - Reclaiming object removes references to other objects
 - May reduce their reference count to zero, so triggering further reclamation



Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI:[10.1007/BFb0017182](https://doi.org/10.1007/BFb0017182)

Reference Counting: Benefits

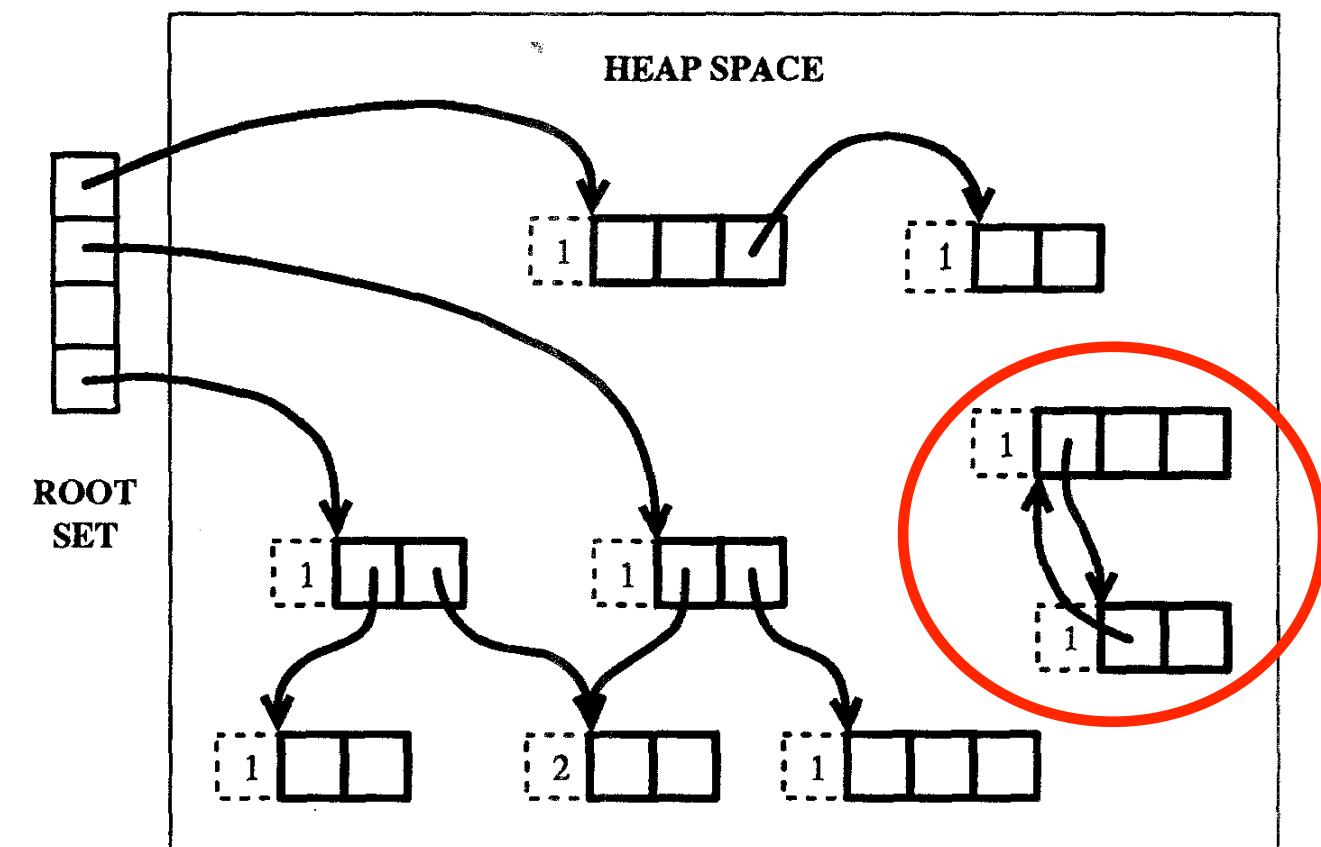
- Incremental operation – memory reclaimed in small bursts
- Predictable and understandable
 - Easy to explain
 - Easy to understand when memory is reclaimed
 - Easy to understand overheads and costs
 - Follows programmer intuition



Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI:[10.1007/BFb0017182](https://doi.org/10.1007/BFb0017182)

Reference Counting: Costs

- Cyclic data structures contain mutual references
 - Objects that reference each other aren't reclaimed, as reference count doesn't go to zero
 - Memory leaks unless cycle explicitly broken; needs programmer action
- Stores reference count alongside each object
 - Maybe also a mutex if concurrent access possible
 - Per-object overhead significant for small objects; wastes memory
- Processor cost of updating references can be significant for short-lived objects



Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI:[10.1007/BFb0017182](https://doi.org/10.1007/BFb0017182)

Reference Counting

- Widely used in scripting languages
 - Python, Ruby, etc.
 - Memory and processor overhead not significant in interpreted runtime
- Used on small scale for systems programming
 - e.g., Objective C runtime on iOS
 - Ease of understanding is important
 - Tends to be for large, long-lived, data – reduces overheads
 - Not typically used in kernel code, high-performance systems

Automatic Memory Management

- Concepts
- Reference counting

Region-based Memory Management

- Concepts and Rationale
- Memory Management in Rust

Region-based Memory Management: Rationale

- Reference counting has relatively high overhead
 - Memory overhead to store the reference count
 - Processor time to update the reference counts
- Garbage collection has unpredictable timing, high overhead → lecture 6
- Manual memory management is too error prone
- Region-based memory management aims for middle ground:
 - Safe, predictable timing – no run-time cost
 - Accepts some impact on application design

Stack-based Memory Management

- Automatic management of stack variables common and efficient:

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

static double pi = 3.14159;

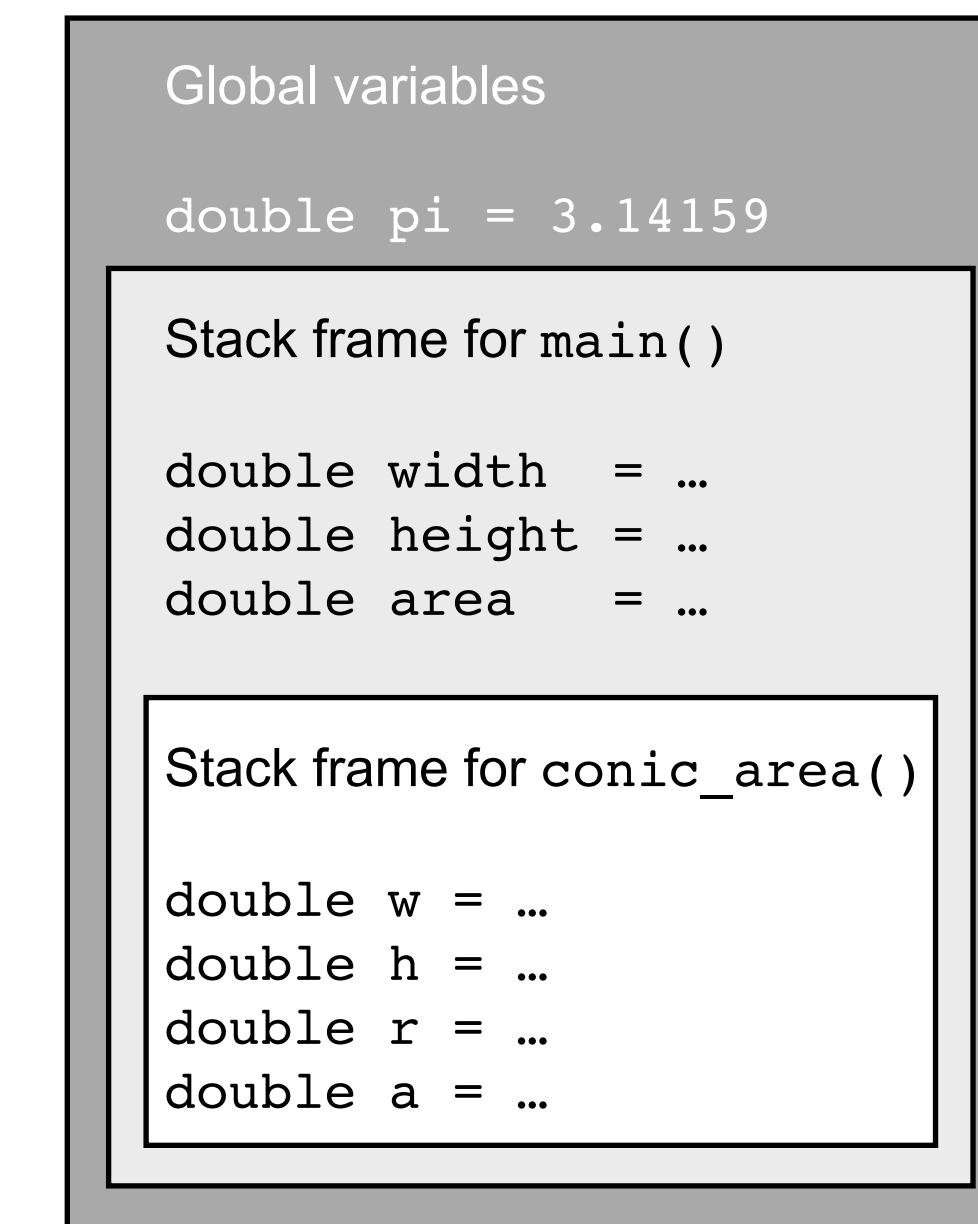
static double conic_area(double w, double h) {
    double r = w / 2.0;
    double a = pi * r * (r + sqrt(h*h + r*r));

    return a;
}

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

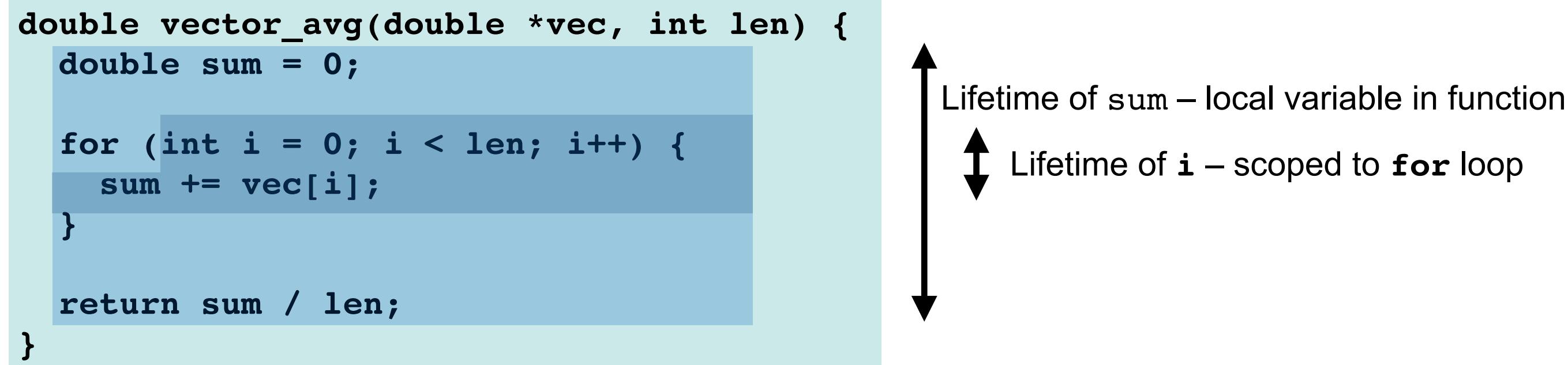
    printf("area of cone = %f\n", area);

    return 0;
}
```



Stack-based Memory Management

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

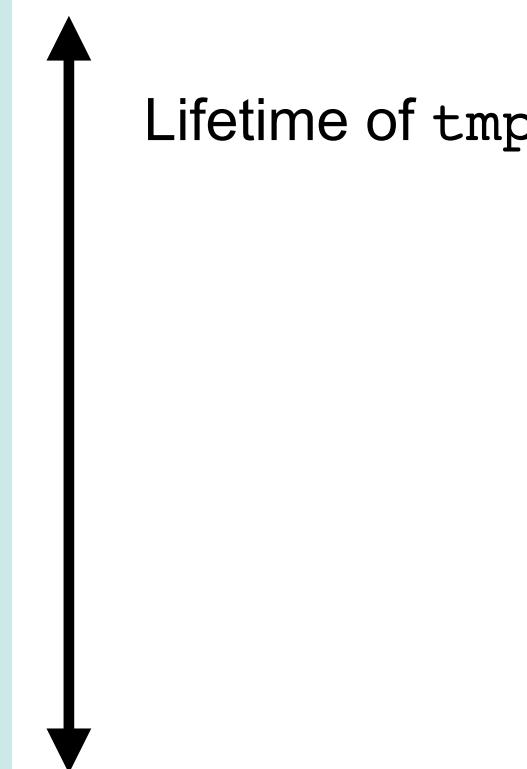


- Variables live within regions, and are deallocated at end of region scope

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;  
}
```



- Local variable **tmp** stored on the stack, freed when function returns
- Memory allocated by **malloc()** is not freed – memory leak

From Stack-to Region-based Memory Management

- Stack-based memory management effective, but limited applicability – can we extend to manage the heap?
 - Track lifetime of data – **values on the stack** and **references to the heap**
 - A **Box<T>** is a value stored on the stack that holds a reference to data of type **T** allocated on the heap:

```
fn main() {  
    let b = Box::new(5);  
    println!("b = {}", b);  
}
```

b is a pointer to heap allocated memory, holding integer value 5

- The **Box<T>** is a normal local variables with lifetime matching the stack frame
- The heap allocated **T** has lifetime matching the **Box<T>** – when the **Box** goes out of scope, the referenced heap memory 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 heap lifetime tied to stack frame lifetime

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

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;

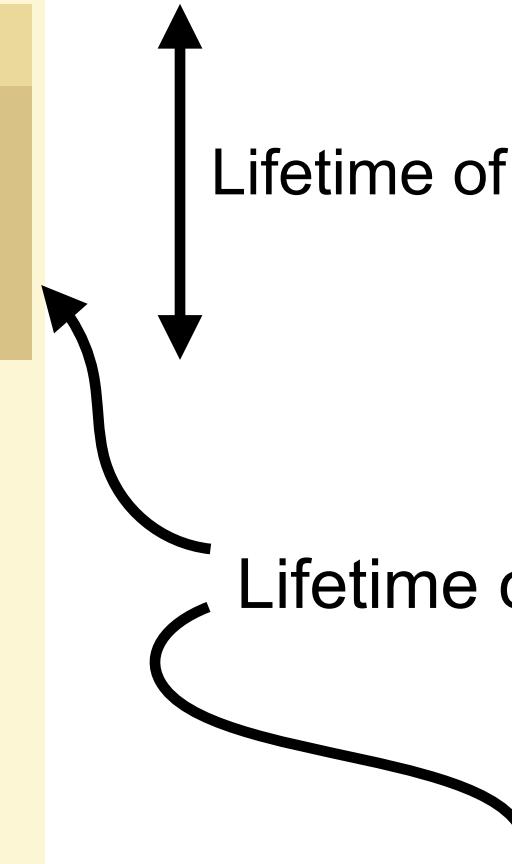
fn area_of_cone(w : f64, h : f64) -> f64 {
    let r = w / 2.0;
    let a = PI * r * (r + (h*h + r*r).sqrt());

    return a;
}

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

    let area = area_of_cone(width, height);

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



Returning Ownership of Data

- Ownership of return value is *moved* to the calling function
 - The value is moved into the calling function's stack frame
 - Original value, in the called function's stack frame, is deallocated
 - Allows us to return a **Box<T>** that references a heap allocated value of type **T**
 - The **Box<T>** is moved, but the referenced **T** on the heap is not
- Variables not returned by a function go out of scope and are reclaimed
 - The heap-allocated **T** is deallocated when the **Box<T>** goes out of scope and is reclaimed
 - i.e., the compiler generates to equivalent of a call to **free()** when the **Box<T>** goes out of scope

No Dangling References

```
fn foo() -> &i32 {  
    let n = 42;  
    &n  
}
```

- Lifetime of local variable ends when function returns
- Can't return a reference to an object that doesn't exist

```
% rustc test.rs  
error[E0106]: missing lifetime specifier  
--> test.rs:1:13  
1 | fn foo() -> &i32 {  
|     ^ expected lifetime parameter  
  
= help: this function's return type contains a borrowed value, but there is no  
= help: value for it to be borrowed from
```

```
int *foo() {  
    int n = 42;  
    return &n;  
}
```

- Equivalent C code will compile but crash at runtime
 - Good compilers give a warning for many, *but not all*, cases

No Use-After-Free

```
use std::mem::drop; // free() equivalent

fn main() {
    let x = "Hello".to_string();
    drop(x);
    println!("{}", x);
}
```

```
error[E0382]: use of moved value: `x`
--> test.rs:6:18
|
5 |     drop(x);
|         - value moved here
6 |     println!("{}", x);
|             ^ value used here after move
|
= note: move occurs because `x` has type `std::string::String`, which does not implement the `Copy` trait
```

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

int main() {
    char *x = malloc(14);
    sprintf(x, "Hello, world!");
    free(x);
    printf("%s\n", x);
}
```

- Similarly – once memory is freed, it cannot be accessed
 - Explicit **drop()** is equivalent of **free()** in C
- Equivalent C program compiles and runs, but has undefined behaviour

Taking Ownership of Data

```
fn consume(mut x : Vec<u32>) {  
    x.push(1);  
}  
  
fn main() {  
    let mut a = Vec::new();  
  
    a.push(1);  
    a.push(2);  
  
    consume(a); // Ownership of a transferred  
                // to consume()  
  
    println!("a.len() = {}", a.len());  
}
```

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

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

Region-based Memory Management

- Concepts and Rationale
- Memory Management in Rust

Resource Management

- **Borrowing Memory**
- Managing Resources

Borrowing Data

```
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 take **references** to data
 - Does **not** move ownership of the data, it borrows it – moves ownership of the reference, not the referenced value
- Functions can also return references to borrowed input parameters
 - The parameters are borrowed from the calling function, so safe to return them to it

Problems with Naïve Borrowing – Iterator Invalidation

```
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  
%
```

- In this example, **borrow()** changes contents of 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**

Safe Borrowing

- Rust has two kinds of pointer:
 - `&T` – shared reference to immutable object
 - `&mut T` – unique reference to mutable object
- The compiler and runtime control reference 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
 - Existence of an `&T` reference to mutable data makes the data immutable
- 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 own 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*

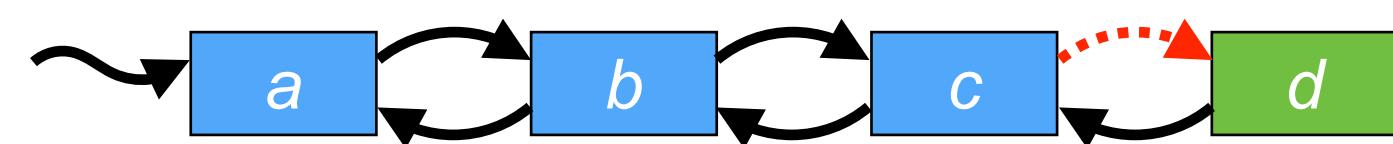
- Enforced by the compiler

Benefits

- Type system tracks ownership, turning run-time bugs into compile-time errors:
 - Prevents 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
 - Generates **exactly the same code** as a correctly written program using `malloc()` and `free()`
 - Timing and memory usage are as predictable as a correct C program
 - Deterministic when memory allocated
 - Deterministic when memory freed

Limitations of Region-based Systems

- Can't express cyclic data structures
 - e.g., can't build a doubly linked list in safe Rust:



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)
 - Fails a run-time exception if there could be a race, rather than preventing it at compile time

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

Region-based Memory Management: 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

Resource Management

- Borrowing Memory
- **Managing Resources**

Resource Management: Deterministic Cleanup

- Rust deterministically frees (“drops”) memory when reference goes out of scope
- Types can implement **Drop** trait to get custom destructors

```
impl Drop for MyType {  
    fn drop(&mut self) {  
        ...  
    }  
}
```

```
pub trait Drop {  
    fn drop(&mut self);  
}
```

Definition of `std::ops::Drop`

- Dropping is deterministic → clean-up resource ownership
 - Garbage collected languages typically give no guarantee when the destructor runs
 - e.g., the **File** class uses custom `drop()` implementation to close the file when it goes out of scope
 - Python has special syntax for this:

```
with open(filename) as file:  
    data = file.read()  
    ...
```

unnecessary in Rust – cleanup happens naturally

Resource Management: Ownership and States

- Use ownership transfer between different types to model resource states
 - **struct**-based state machine → lecture 4

```
impl UnauthenticatedConnection {  
    fn login(self, c: Credentials) -> Result<AuthenticatedConnection, ...> {  
        ...  
    }  
  
    fn disconnect(self) -> NotConnected {  
        ...  
    }  
}
```

- Manage the different states of a resource
- Make illegal operations compile time errors
- See also:
<https://blog.systems.ethz.ch/blog/2018/a-hammer-you-can-only-hold-by-the-handle.html>

Memory and Resource Management

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

Garbage Collection

Advanced Systems Programming (H/M)
Lecture 6



Rationale

- Region-based memory management (→ lecture 5) is novel, trades program complexity for predictable resource management
- Garbage collection widely implemented, but less predictable
- Need to understand garbage collector operation to understand the performance-complexity trade-off

Lecture Outline

- Garbage collection algorithms
 - Mark-sweep
 - Mark-compact
 - Copying collectors
 - Generational collectors
 - Incremental collectors
- Real-time garbage collection
- Practical factors

P. R. Wilson, “Uniprocessor garbage collection techniques”, Proceedings of the International Workshop on Memory Management, St. Malo, France, September 1992. DOI: [10.1007/BFb0017182](https://doi.org/10.1007/BFb0017182)

Uniprocessor Garbage Collection Techniques

Paul R. Wilson

University of Texas
Austin, Texas 78712-1188 USA
(wilson@cs.utexas.edu)

Abstract. We survey basic garbage collection algorithms, and variations such as incremental and generational collection. The basic algorithms include reference counting, mark-sweep, mark-compact, copying, and treadmill collection. *Incremental* techniques can keep garbage collection pause times short, by interleaving small amounts of collection work with program execution. *Generational* schemes improve efficiency and locality by garbage collecting a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects.

1 Automatic Storage Reclamation

Garbage collection is the automatic reclamation of computer storage [Knu69, Coh81, App91]. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a “free” or “dispose” statement, garbage collected systems free the programmer from this burden. The garbage collector’s function is to find data objects¹ that are no longer in use and make their space available for reuse by the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a “dangling pointer” into a deallocated object.

This paper is intended to be an introductory survey of garbage collectors for uniprocessors, especially those developed in the last decade. For a more thorough treatment of older techniques, see [Knu69, Coh81].

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when *other* modules are not interested in a particular object.

¹ We use the term *object* loosely, to include any kind of structured data record, such as Pascal records or C structs, as well as full-fledged objects with encapsulation and inheritance, in the sense of object-oriented programming.

Basic Garbage Collection

- Mark-sweep
- Mark-compact
- Copying collectors

Garbage Collection

- Avoid problems of reference counting and complexity of compile-time ownership tracking via **garbage collection**
 - Explicitly trace through allocated objects, recording which are in use; dispose of unused objects
 - Moves garbage collection to be a separate phase of the program's execution, rather than an integrated part of an objects lifecycle
 - Operation of the program (the **mutator**) and the garbage collector is interleaved
- Many tracing garbage collection algorithms exist:
 - Basic garbage collectors
 - Mark-sweep collectors
 - Mark-compact collectors
 - Copying collectors
 - Generational garbage collectors

Mark-Sweep Collectors (1/4)

- Simplest automatic garbage collection scheme
- Two phase algorithm
 - Distinguish live objects from garbage (*mark*)
 - Reclaim the garbage (*sweep*)
 - Non-incremental: program is paused to perform collection when memory becomes tight

Mark-Sweep Collectors (2/4)

- The **marking phase**: distinguishing live objects
 - Determine the **root set** of objects, comprising:
 - Any global variables
 - Any variable allocated on the stack, in any existing stack frame
 - Traverse the object graph starting at the root set to find other reachable objects
 - Starting from the root set, follow pointers to other objects
 - Follow every pointer in every object to systematically find all reachable objects
 - May proceed either breadth-first or depth-first
 - A cycle of objects that reference each other, but are not reachable from the root set, will not be marked
 - Mark reachable objects as alive
 - Set a bit in the object header, or in some separate table of live objects, to indicate that the object is reachable
 - Stop traversal at previously seen objects to avoid looping forever

Mark-Sweep Collectors (3/4)

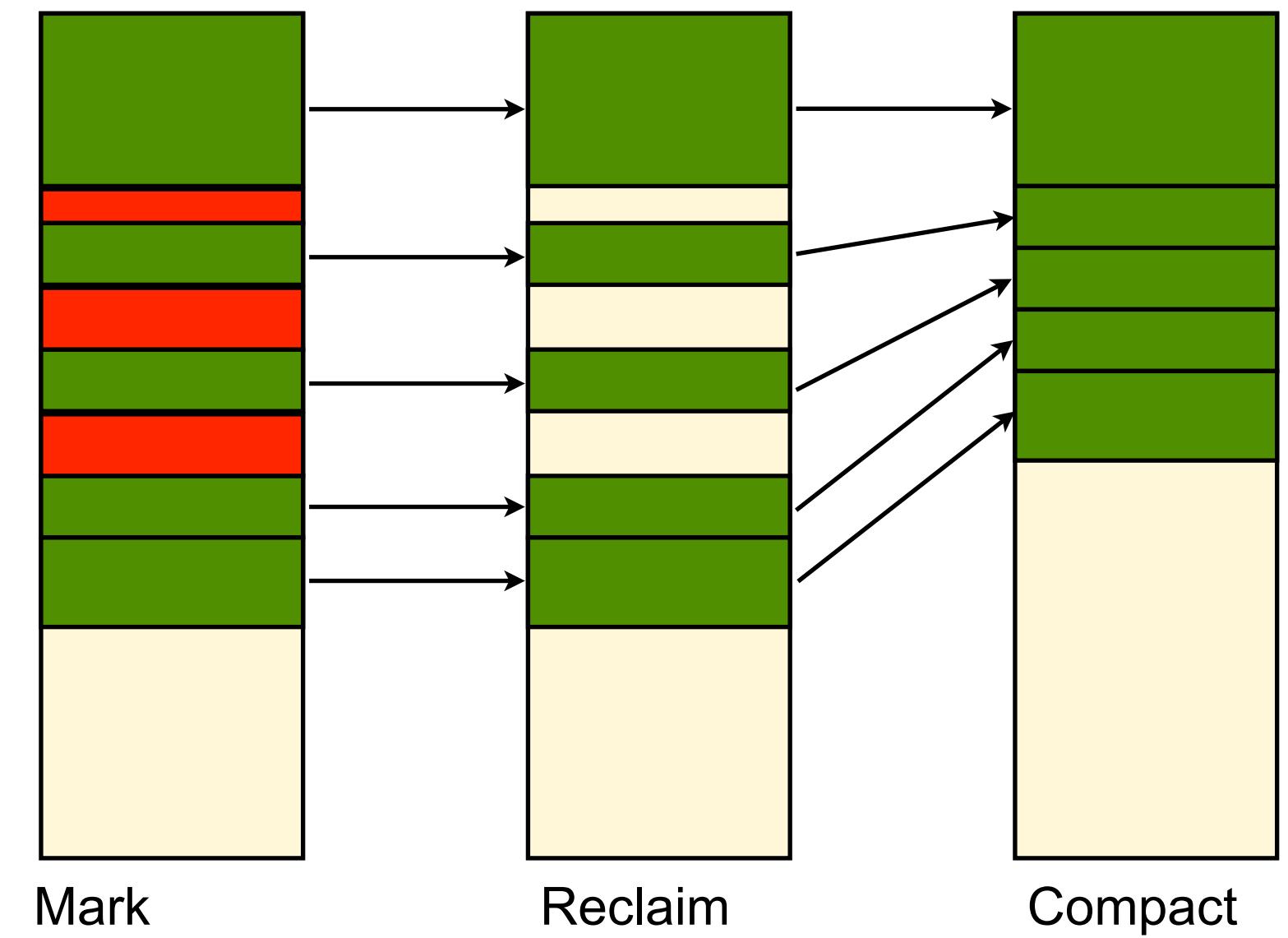
- The **sweep phase**: reclaiming objects that no longer live
 - Pass through entire heap once, examining each object for liveness
 - If marked as alive, keep the object
 - Otherwise, free the memory and reclaim the object's space
- When objects are reclaimed, their memory is marked as available
 - The system maintains a **free list** of blocks of unused memory
 - New objects are allocated in now unused memory if they fit; or in not-yet-used memory elsewhere on the heap
 - Fragmentation is a potential concern – but no worse than using `malloc()`/`free()`

Mark-Sweep Collectors (4/4)

- Mark-sweep collectors are simple, but inefficient:
 - Garbage collection is slow and has unpredictable duration
 - Program is stopped while the collector runs
 - Time to collect garbage is unpredictable, and depends on the number of live objects (time for the marking phase) and size of the heap (time to sweep up unused objects)
 - Unlike reference counting, mark-sweep garbage collection is slower if the program has lots of memory allocated
 - Garbage collection has no locality of reference
 - Passing through the entire heap in unpredictable order disrupts operation of cache and virtual memory subsystem
 - Objects located where they fit, rather than where maintains locality of reference
 - Fragmentation of free space is a concern
 - Since objects are not moved, space may become fragmented, making it difficult to allocate large objects even though space available overall

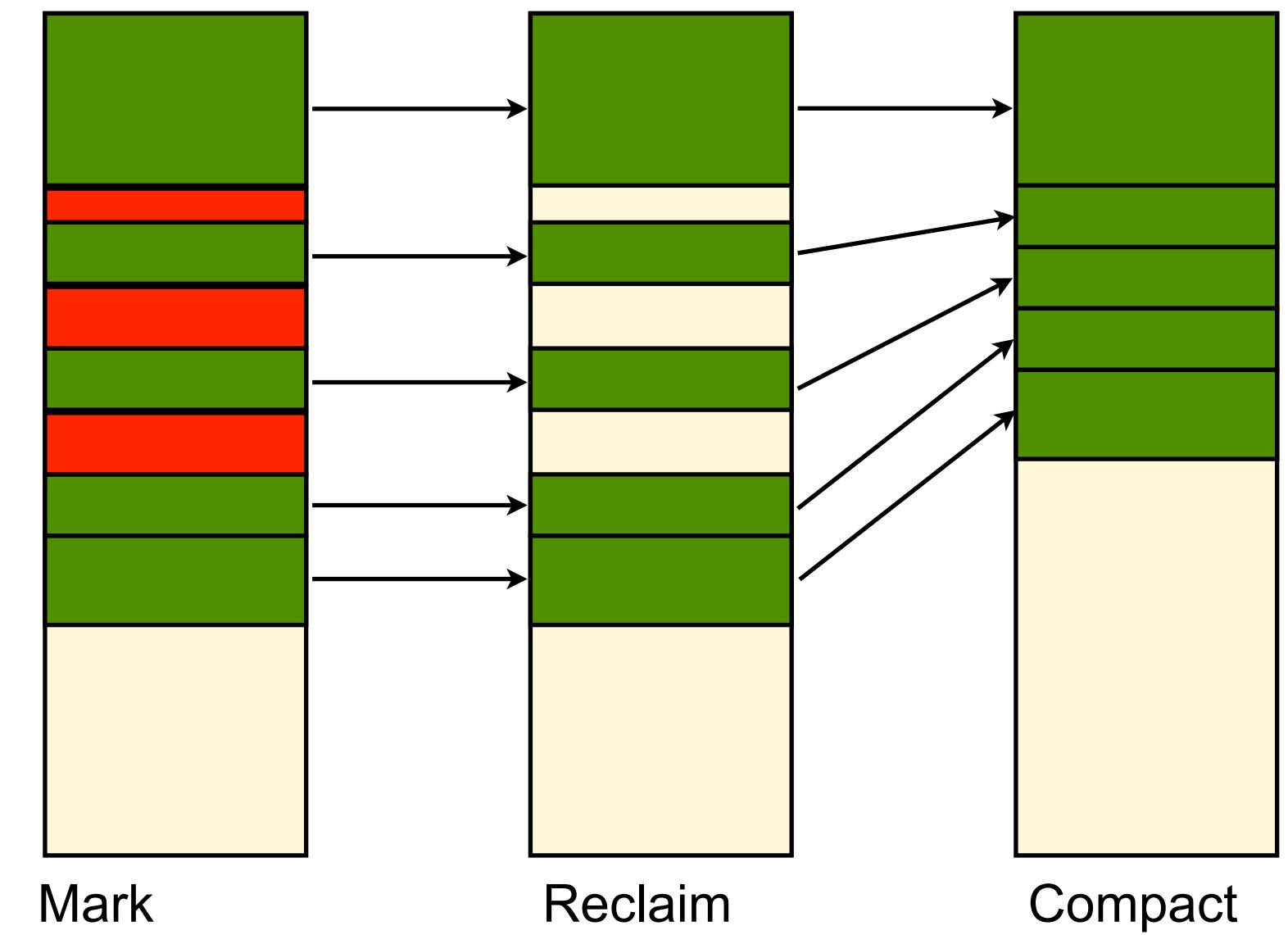
Mark-Compact Collectors (1/2)

- Goal: solve fragmentation problems and speed-up allocation, compared to mark-sweep collectors
- Three logical phases:
 - Traverse object graph, **mark** live objects
 - **Reclaim** unreachable objects
 - **Compact** live objects, moving them to leave contiguous free space
- Reclaiming and compacting memory can be done in one pass, but still touches the entire address space



Mark-Compact Collectors (2/2)

- Advantages:
 - Solves fragmentation problems – all free space is in one contiguous block
 - Allocation is very fast – always allocating from the start of the free block, so allocation is just incrementing pointer to start of free space and returning previous value
- Disadvantages:
 - Collection is slow, due to moving objects in memory, and time taken is unpredictable
 - Collection has poor locality of reference
 - Collection is complex – needs to update all pointers to moved objects

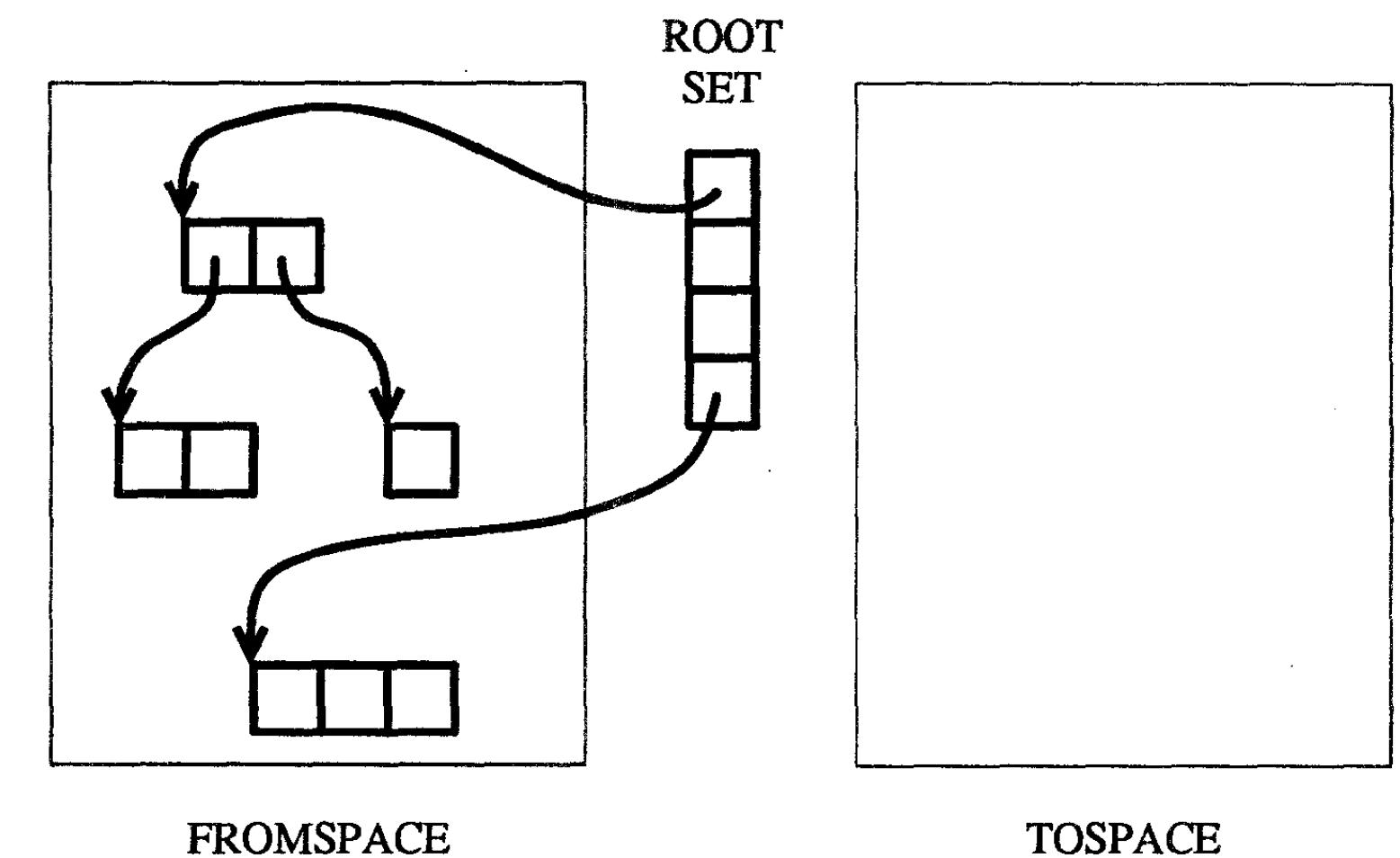


Copying Collectors (1/5)

- Copying collectors integrate traversal (marking) and copying phases into one pass
 - All the live data is copied into one region of memory
 - All the remaining memory contains garbage, or has not yet been used
- Similar to mark-compact, but more efficient
- Time taken to collect is proportional to the number of live objects

Copying Collectors (2/5)

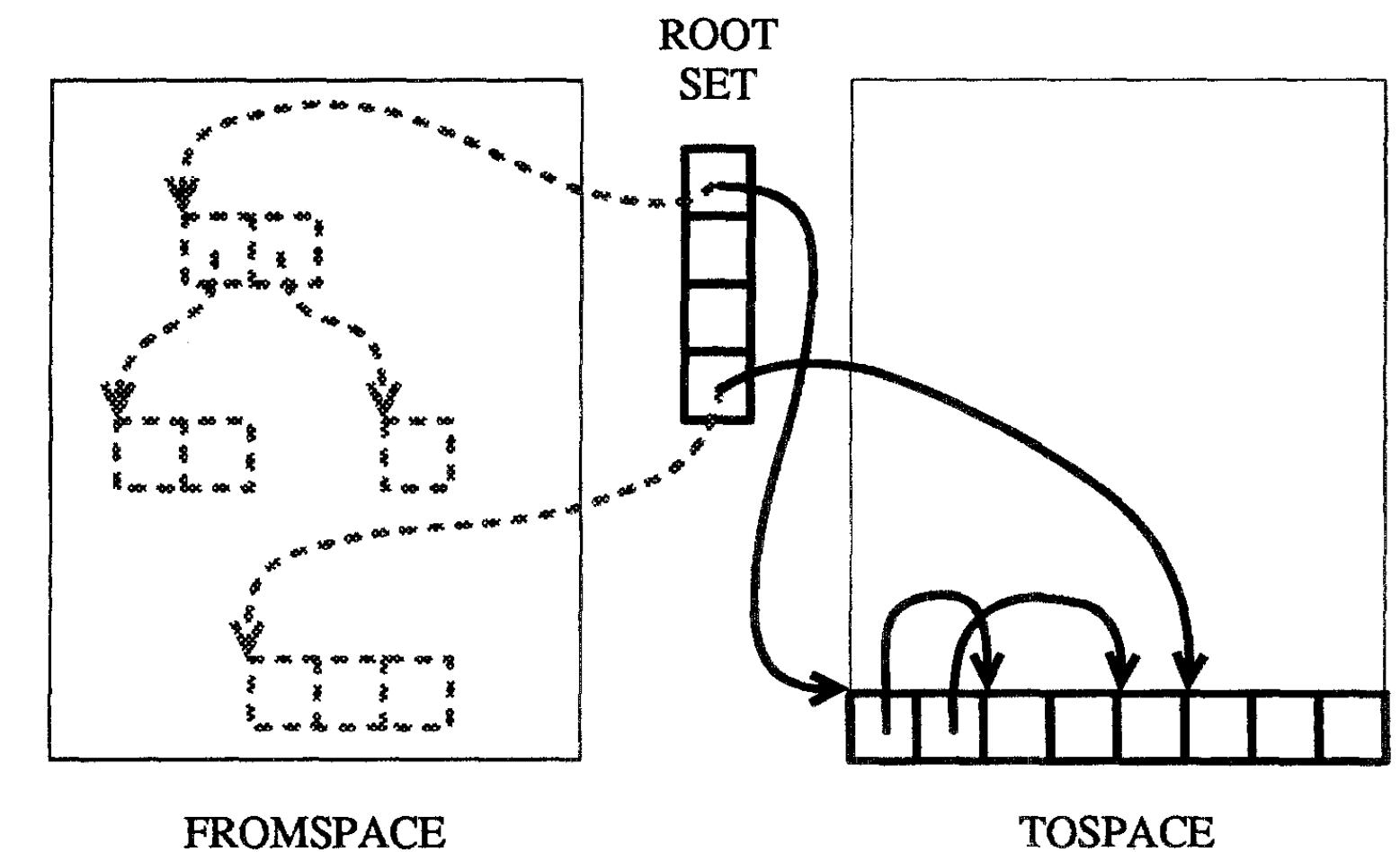
- Stop-and-copy using semispaces:
 - Divide the heap into two halves, each one a contiguous block of memory
 - Allocations made linearly from one half of the heap only
 - Memory is allocated contiguously, so allocation is fast (as in the mark-compact collector)
 - No problems with fragmentation when allocating data of different sizes
 - When an allocation is requested that won't fit into the active half of the heap, a collection is triggered



Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI 10.1007/BFb0017182

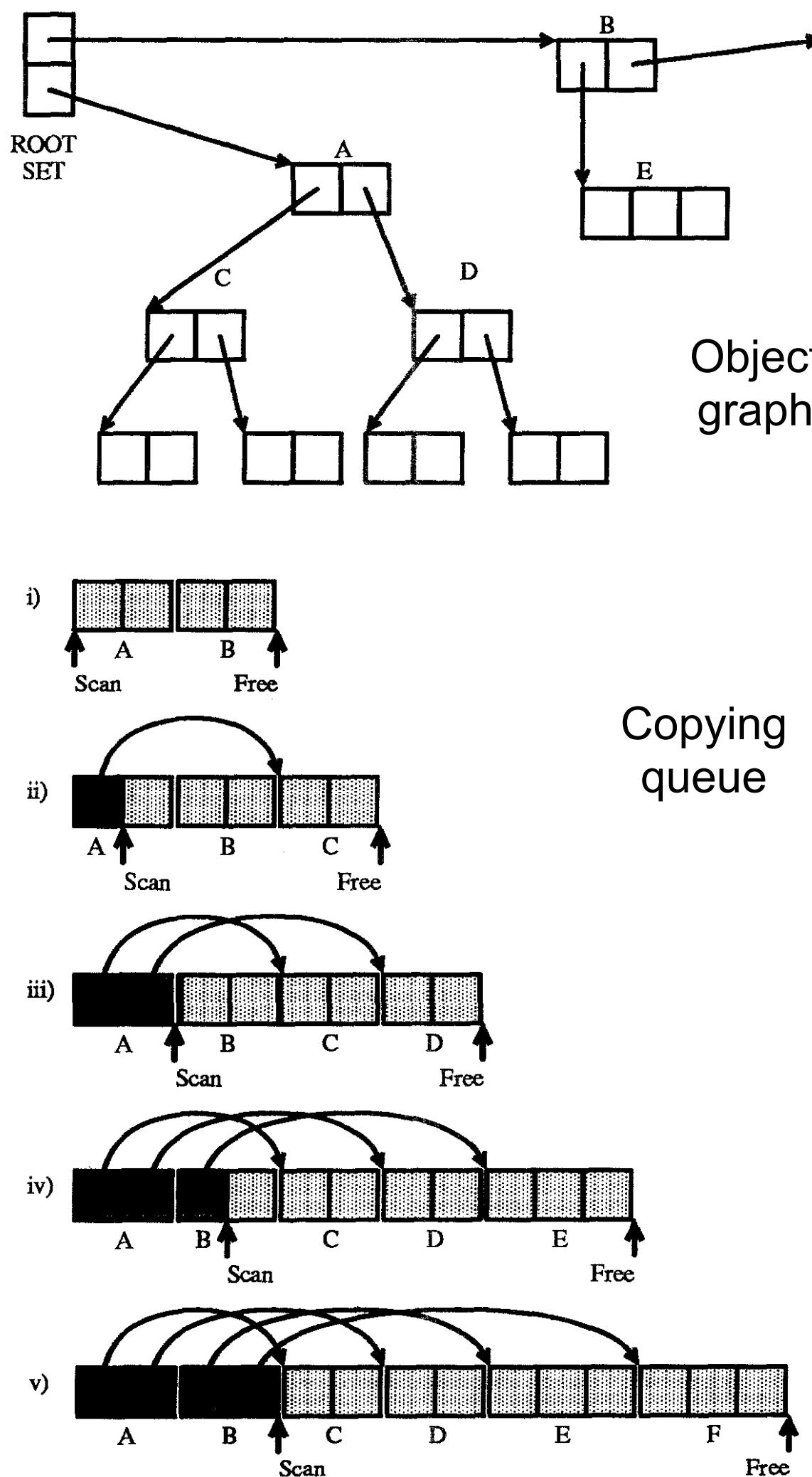
Copying Collectors (3/5)

- Stop-and-copy using semispaces:
 - Collection stops execution of the program
 - A pass is made through the active space, and all live objects are copied to the other half of the heap
 - Cheney algorithm is commonly used to make the copy in a single pass
 - Anything not copied is unreachable, and is simply ignored (and will eventually be overwritten by a later allocation phase)
 - The program is then restarted, using the other half of the heap as the active allocation region
 - The role of the two parts of the heap (the two semispaces) reverses each time a collection is triggered



Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI 10.1007/BFb0017182

Copying Collectors (4/5)



- The Cheney Algorithm: breadth-first copying
 - A queue is created, to hold the set of live objects to be copied
 - The **root set** of objects, comprising global variables and all stack allocated variables, is found and added to the queue
 - Objects in the queue are examined in turn:
 - Any unprocessed objects they reference are added to end of the queue
 - The object in the queue is then copied into the other semispace, and the original is marked as having been processed (pointers are updated as the copy is made)
 - Once the end of the queue is reached, all live objects have been copied

Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI 10.1007/BFb0017182

Copying Collectors (5/5)

- Efficiency of copying collectors:
 - Time taken for garbage collection depends on the amount of data copied, which depends on the number of live objects
 - Collection only happens when a semispace is full
- **If most objects die young**, can trade-off collection time vs. memory usage by increasing the size of the semispaces
 - A larger semispace takes longer to fill, so increases the how long objects need to live before they're copied
 - If most objects die young, most aren't copied
 - Uses more memory, but spends less time copying data

Summary: Basic Garbage Collection

- Mark-sweep, mark-compact, and copying collectors have similar cost:
 - **Differ in where the cost is paid:** at time of allocation or time of collection; in memory usage or in processing time
- The mark-compact and copying algorithms move data, so cannot be used with languages that cannot unambiguously identify pointers
 - Can't move an object, if you can't be sure all pointers to it have been updated

Basic Garbage Collection

- Mark-sweep
- Mark-compact
- Copying collectors

Generational and Incremental Garbage Collection

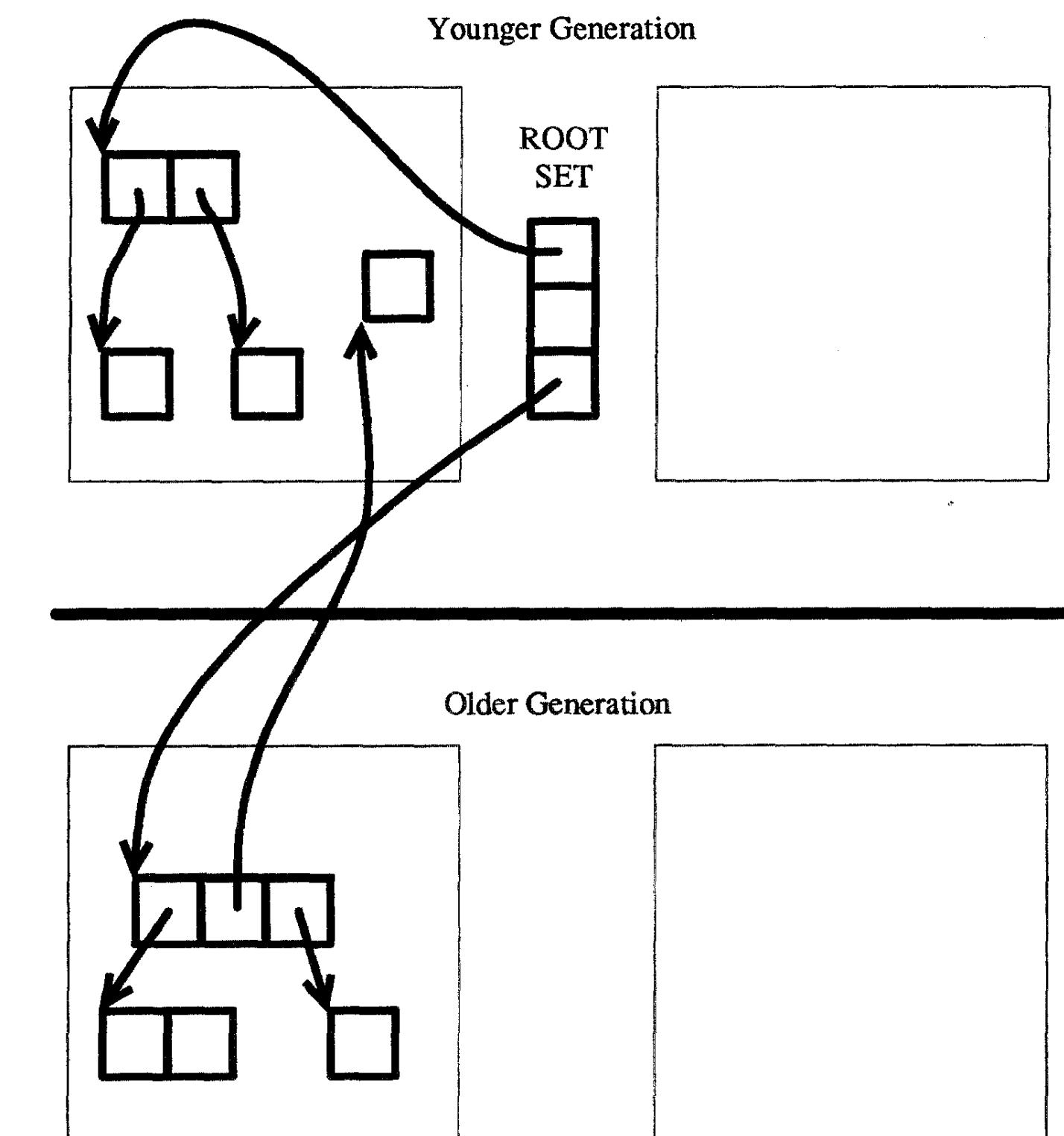
- Object Lifetimes
- Copying Generational Collectors
- Incremental Garbage Collection

Object Lifetimes

- Most objects have short time; a small percentage live much longer
 - This seems to be generally true, no matter what programming language is considered, across numerous studies
 - Although, obviously, different programs and different languages produce varying amount of garbage
- Implications:
 - When the garbage collector runs, live objects will be in a minority
 - Statistically, the longer an object has lived, the longer it is likely to live
 - Can we design a garbage collector to take advantage?

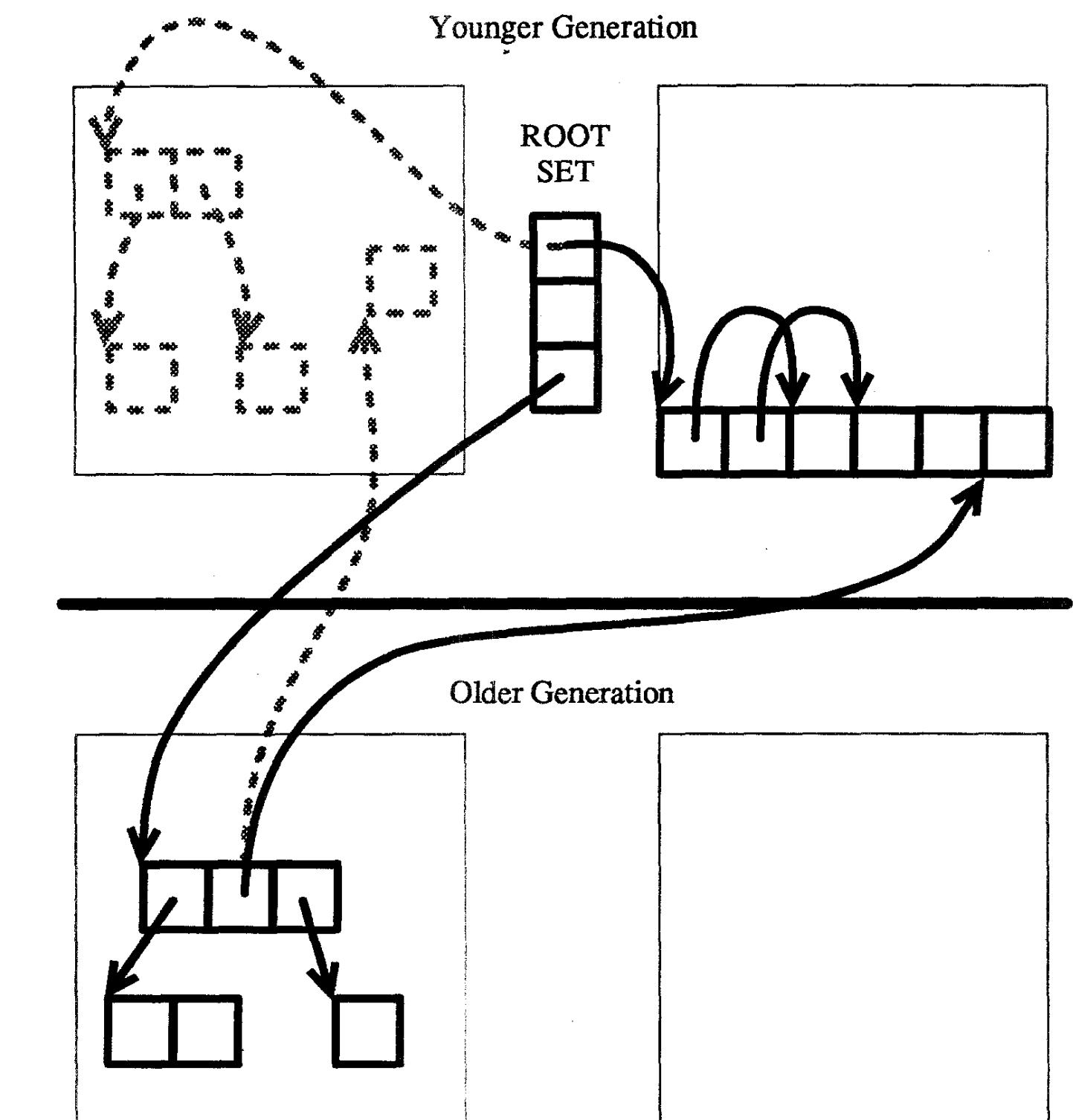
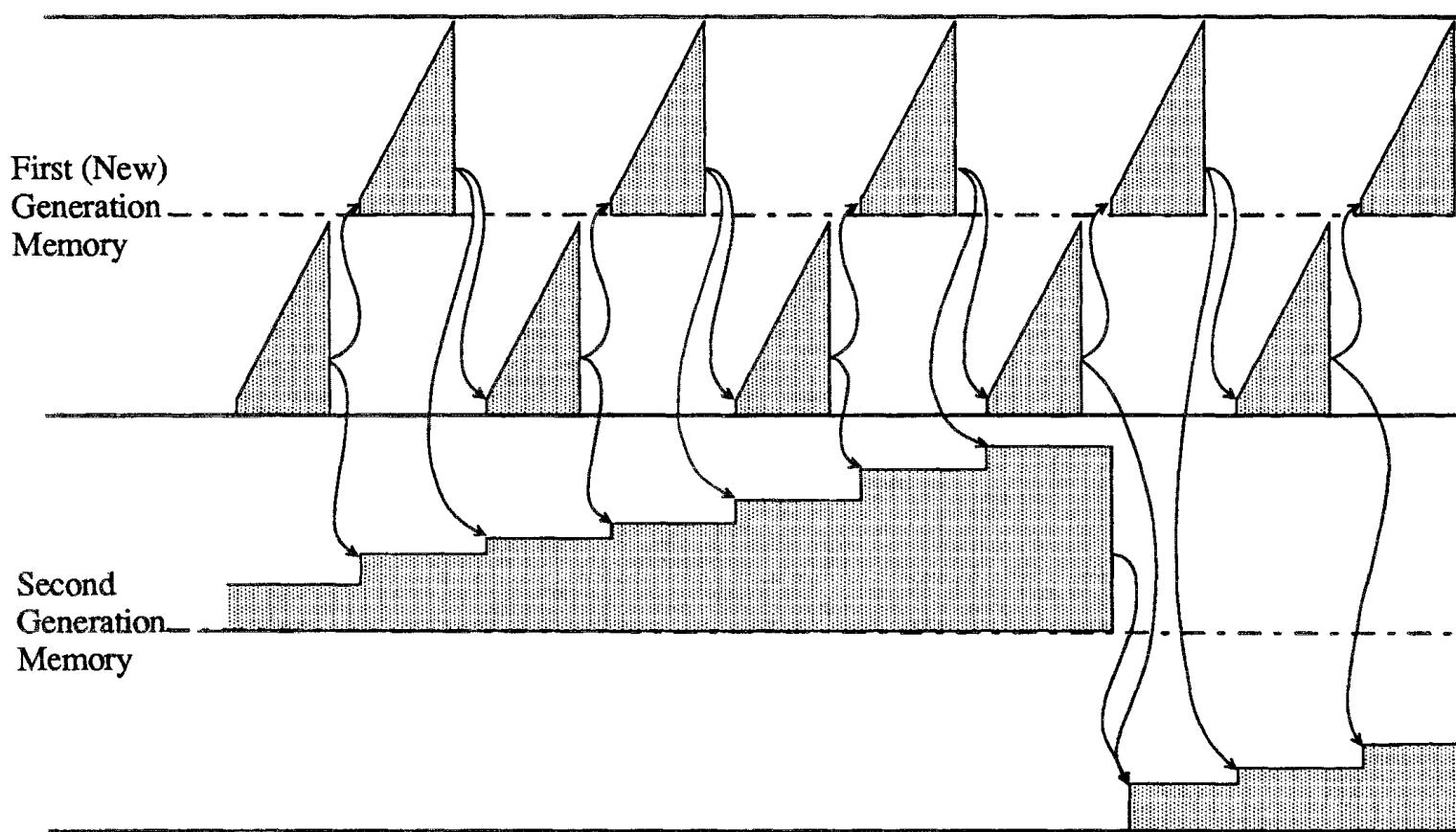
Copying Generational Collectors (1/4)

- In a generational garbage collector, the heap is split into regions for long-lived and young objects
 - Regions holding young objects are garbage collected more frequently
 - Objects are moved to the region for long-lived objects if they're still alive after several collections
 - More sophisticated approaches may have multiple generations, although the gains diminish rapidly with increasing numbers of generations
 - Example: stop-and-copy using semispaces with two generations
 - All allocations occurs in the younger generation's region of the heap
 - When that region is full, collection occurs as normal
 - ...



Copying Generational Collectors (2/4)

- ...
- Objects are tagged with the number of collections of the younger generation they have survived; if they're alive after some threshold, they're copied to the older generation's space during collection
- Eventually, the older generation's space is full, and is collected as normal



- Note: not to scale: older generations are generally much larger than the younger, as they're collected much less often

Copying Generational Collectors (3/4)

- Young generation must be collected independent of long-lived generation
- But – there may be references between generations
 - References from young objects to long-lived objects
 - Straight-forward – most young objects die before the long-lived objects are collected
 - Treat the younger generation objects as part of the root set for the long-lived generation, when collection of the long-lived generation is needed
 - References from long-lived objects to young objects:
 - Problematic, since requires scan of long-lived generation to detect
 - Maybe use indirection table (“pointers-to-pointers”) for references from long-lived generation to young generation
 - The indirection table forms part of the root set of the younger generation
 - Moving objects in younger generation requires updating indirection table, but not long-lived objects
 - Long-lived objects are collected infrequently, and may keep younger objects alive longer than expected

Copying Generational Collectors (4/4)

- Variations on copying generational collectors are widely used
 - E.g., the HotSpot JVM uses a generational garbage collector
- Copying generational collectors are efficient:
 - Cost of collection is generally proportional to number of live objects
 - Most objects don't live long enough to be collected; those that do are moved to a more rarely collected generation
 - Longer-lived generation must eventually be collected; this can be very slow

Incremental Garbage Collection (1/5)

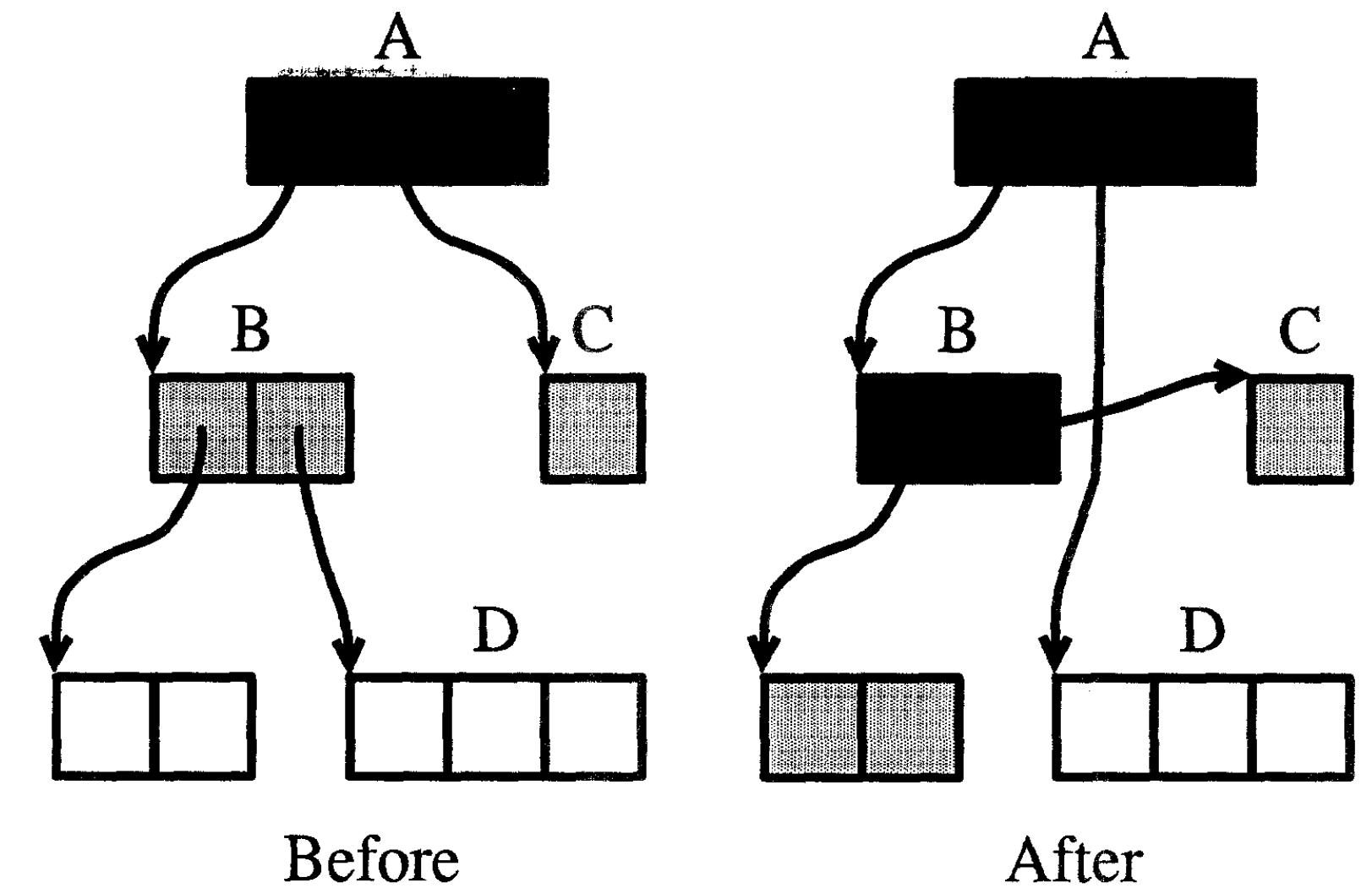
- Preceding discussion has assumed the collector “stops-the-world” when it runs
 - Problematic for interactive or real-time applications
 - Desire a collector that can operate incrementally
 - Interleave small amounts of garbage collection with small runs of program execution
 - Implication: the garbage collector can’t scan the entire heap when it runs; must scan a fragment of the heap each time
 - Problem: the program (the “mutator”) can change the heap between runs of the garbage collector
 - Need to track changes made to the heap between garbage collector runs; be conservative and don’t collect objects that might be referenced – can always collect on the next complete scan

Incremental Garbage Collection (2/5)

- Tricolour marking: each object is labelled with a colour:
 - White – not yet checked
 - Grey – live, but some direct children not yet checked
 - Black – live
- Basic incremental collector operation:
 - Garbage collection proceeds with a waveform of grey objects, where the collector is checking them, or objects they reference, for liveness
 - Black objects behind are behind the waveform, and are known to be live
 - Objects ahead of the waveform, not yet reached by the collection, are white; anything still white once all objects have been traced is garbage
 - No direct pointers from black objects to white – any program operation that will create such a pointer requires coordination with the collector

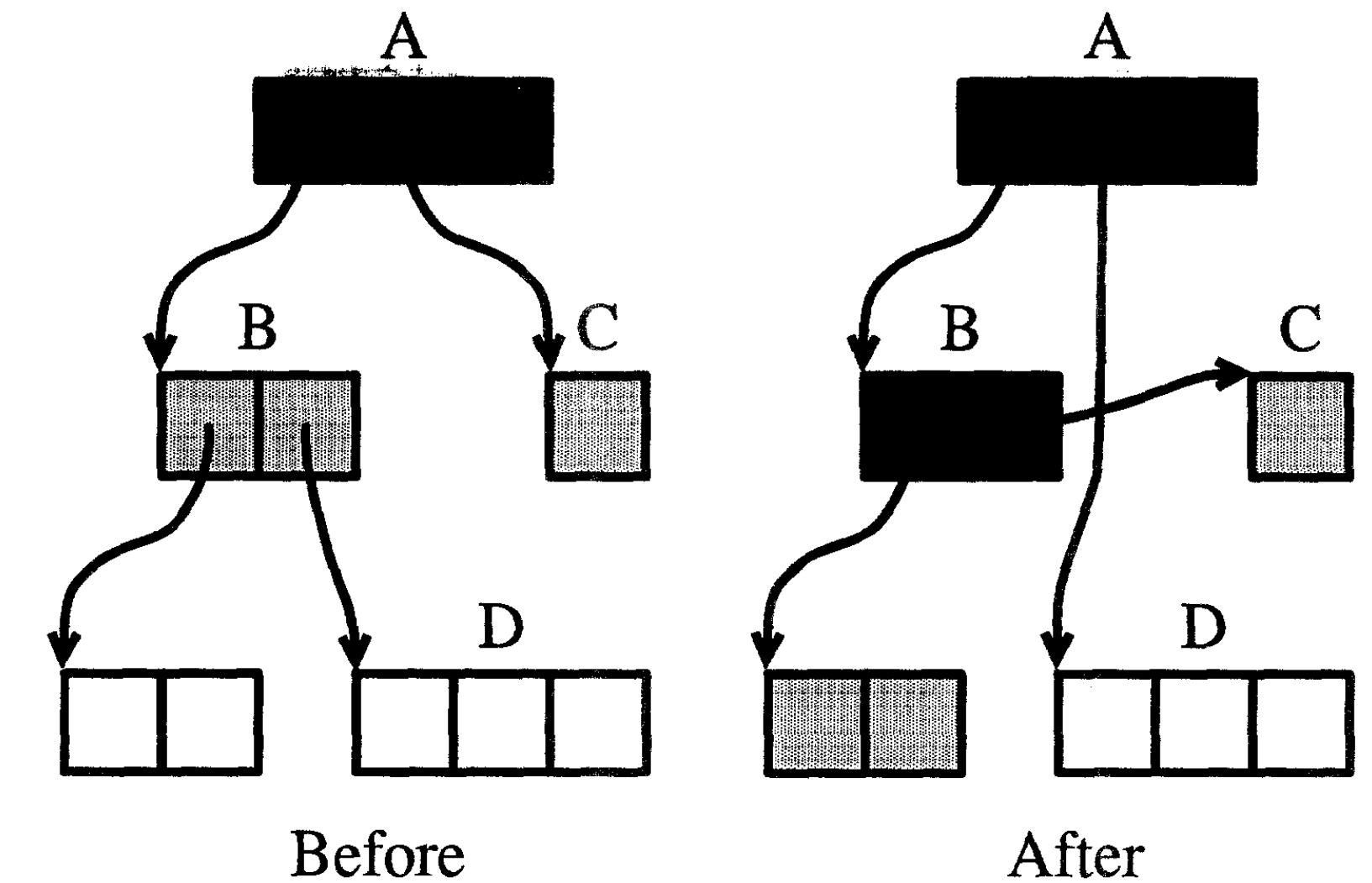
Incremental Garbage Collection (3/5)

- Program and collector must coordinate
 - Garbage collector runs
 - Object A scanned, known to be live → black
 - Objects B and C are reachable via A, and are live, but some of their children have not been scanned → grey
 - Object D not checked → white
 - Program runs, and swaps the pointers from A→C and B→D such that A→D and B→C
 - This creates a pointer from black to white
 - Program must now coordinate with the collector, else collection will continue, marking object B black and its children grey, but D will not be reached since children of A have already been scanned



Incremental Garbage Collection (4/5)

- Coordination strategies:
 - Read barrier: trap attempts by the program to read pointers to white objects, colour those objects grey, and then let program continue
 - Makes it impossible for the program to get a pointer to a white object, so it cannot make a black object point to a white
 - Write barrier: trap attempts to change pointers from black objects to point to white objects
 - Either then re-colour the black object as grey, or re-colour the white object being referenced as grey
 - The object coloured grey is moved onto the list of objects whose children must be checked



Incremental Garbage Collection (5/5)

- Many variants on read- and write-barrier tricolour algorithms
 - Performance trade-off differs depending on hardware characteristics, and on the way pointers are represented
 - Write barrier generally cheaper to implement than read barrier, as writes are less common in most code
- There is a balance between collector operation and program operation
 - If the program tries to create too many new references from black to white objects, requiring coordination with the collector, the collection may never complete
 - Resolve by forcing a complete stop-the-world collection if free memory is exhausted, or after a certain amount of time

Generational and Incremental Garbage Collection

- Object Lifetimes
- Copying Generational Collectors
- Incremental Garbage Collection

Practical Factors

- Real-time Garbage Collection
- Memory Overheads
- Interaction with Virtual Memory
- Garbage Collection for Weakly-Typed Languages

Real-time Garbage Collection

- Real-time collectors built from incremental collectors
 - Schedule an incremental collector as a periodic task
 - Runtime allocated determines amount of garbage that can be collected in each period
 - The amount of garbage that can be collected can be measured: how fast can the collector scan memory (and copy objects, if a copying collector)
 - The programmer must bound the amount of garbage generated to within the capacity of the collector
 - Hard real-time systems **must** always stay within the bounds of the collector
 - Soft real-time systems meet statistical bounds

A Real-time Garbage Collector with Low Overhead and Consistent Utilization

David F. Bacon
dfb@watson.ibm.com

Perry Cheng
perryche@us.ibm.com

V.T. Rajan
vtrajan@us.ibm.com

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT
Now that the use of garbage collection in languages like Java is becoming widely accepted due to the safety and software engineering benefits it provides, there is significant interest in applying garbage collection to hard real-time systems. Past approaches have generally suffered from one of two major flaws: either they were not provably real-time, or they imposed large space overheads to meet the real-time bounds. We present a mostly non-moving, dynamically defragmenting collector that overcomes both of these limitations: by avoiding copying in most cases, space requirements are kept low; and by fully incrementalizing the collector we are able to meet real-time bounds. We implemented our algorithm in the Jikes RVM and show that at real-time resolution we are able to obtain mutator utilization rates of 45% with only 1.6–2.5 times the actual space required by the application, a factor of 4 improvement in utilization over the best previously published results. Defragmentation causes no more than 4% of the traced data to be copied.

General Terms
Algorithms, Languages, Measurement, Performance

Categories and Subject Descriptors
C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems; D.3.2 [Programming Languages]: Java; D.3.4 [Programming Languages]: Processors—Memory management (garbage collection)

Keywords
Read barrier, defragmentation, real-time scheduling, utilization

1. INTRODUCTION
Garbage collected languages like Java are making significant inroads into domains with hard real-time concerns, such as automotive command-and-control systems. However, the engineering and product life-cycle advantages consequent from the simplicity of

programming with garbage collection remain unavailable for use in the core functionality of such systems, where hard real-time constraints must be met. As a result, real-time programming requires the use of multiple languages, or at least (in the case of the Real-Time Specification for Java [9]) two programming models within the same language. Therefore, there is a pressing practical need for a system that can provide real-time guarantees for Java without imposing major penalties in space or time.

We present a design for a real-time garbage collector for Java, an analysis of its real-time properties, and implementation results that show that we are able to run applications with high mutator utilization and low variance in pause times.

The target is uniprocessor embedded systems. The collector is therefore concurrent, but not parallel. This choice both complicates and simplifies the design: the design is complicated by the fact that the collector must be interleaved with the mutators, instead of being able to run on a separate processor; the design is simplified since the programming model is sequentially consistent.

Previous incremental collectors either attempt to avoid overhead and complexity by using a non-copying approach (and are therefore subject to potentially unbounded fragmentation), or attempt to prevent fragmentation by performing concurrent copying (and therefore require a minimum of a factor of two overhead in space, as well as requiring barriers on reads and/or writes, which are costly and tend to make response time unpredictable).

Our collector is unique in that it occupies an under-explored portion of the design space for real-time incremental collectors: it is a *mostly non-copying* hybrid. As long as space is available, it acts like a non-copying collector, with the consequent advantages. When space becomes scarce, it performs defragmentation with limited copying of objects. We show experimentally that such a design is able to achieve low space and time overhead, and high and consistent mutator CPU utilization.

In order to achieve high performance with a copying collector, we have developed optimization techniques for the Brooks-style read barrier [10] using an “eager invariant” that keeps read barrier overhead to 4%, an order of magnitude faster than previous software read barriers.

Our collector can use either time- or work-based scheduling. Most previous work on real-time garbage collection, starting with Baker’s algorithm [5], has used work-based scheduling. We show both analytically and experimentally that time-based scheduling is superior, particularly at the short intervals that are typically of interest in real-time systems. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The paper is organized as follows: Section 2 describes previ-

POPL’03, January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright © 2003 ACM 1-58113-628-5/03/0001 \$5.00.

285

Bacon *et al.*, “A real-time garbage collector with low overhead and consistent utilization”. ACM Symposium on Principles of Programming Languages, New Orleans, LA, USA, January 2003. DOI: [10.1145/604131.604155](https://doi.org/10.1145/604131.604155)

Memory Overheads

- Garbage collection trades ease-of-use for predictability and overhead
- Garbage collected programs will use significantly more memory than **correctly written** programs with manual memory management
 - Many copying collectors maintain two semispaces, so double memory usage
 - But – many programs with manual memory management are **not correct**

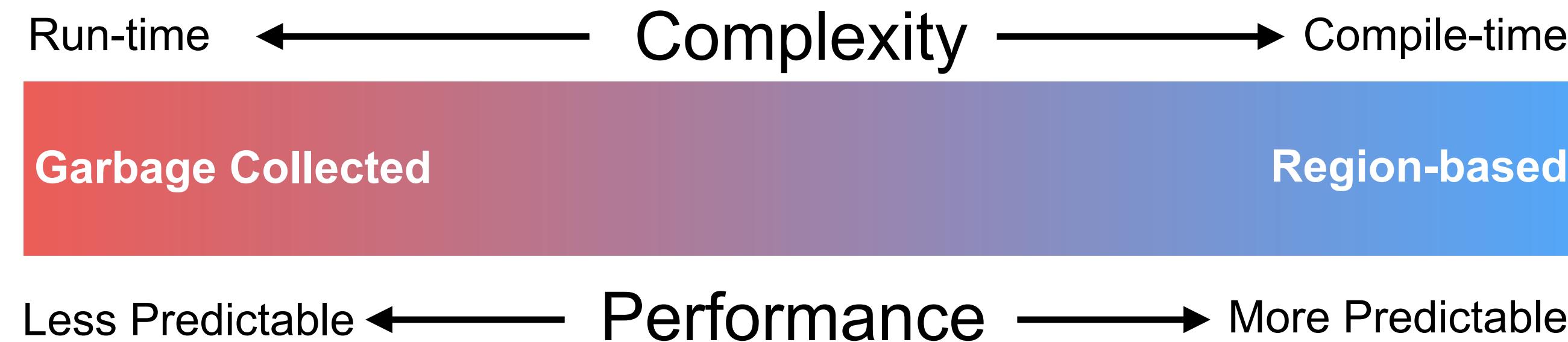
Interaction with Virtual Memory

- Virtual memory subsystems page out unused data in an LRU manner
- Garbage collector scans objects, paging data back into memory
- Leads to thrashing if the working set of the garbage collector larger than memory
 - Open research issue: combining virtual memory with garbage collector

Garbage Collection for Weakly-typed Languages

- Collectors rely on being able to identify and follow pointers, to determine what is a live object – they rely on strongly-typed languages
- Weakly typed languages, such as C, can cast any integer to a pointer, and perform pointer arithmetic
 - Implementation-defined behaviour, since pointers and integers are not guaranteed to be the same size
 - Difficult, but not impossible, to write a garbage collector for C:
 - Need to be conservative: any memory that might be a pointer must be treated as one
 - Boehm-Demers-Weiser collector can be used for C and C++ (<http://www.hboehm.info/gc/>) – works for strictly conforming ANSI C code, but beware that much code is not conforming
 - Other weakly typed languages may suffer from similar problems
 - Strongly typed, but dynamic, languages, such as Python, not an issue

Memory Management Trade-offs



- Rust pushes memory management complexity onto the programmer
 - Predictable run-time performance, low run-time overheads
 - Uniform resource management framework, including memory
 - Limits the programs that may be expressed – matches common patterns in good C code
- Garbage collection imposes run-time costs and complexity, simpler for programmer

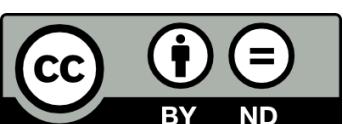
Summary

- Garbage collection
 - Mark-sweep
 - Mark-compact
 - Copying collectors
 - Generational algorithms
 - Incremental algorithms
 - Real-time garbage collection
 - Practical factors



Concurrency

Advanced Systems Programming (H/M)
Lecture 7



Colin Perkins | <https://csperrkins.org/> | Copyright © 2020 University of Glasgow | 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

- Implications of Multicore Systems
 - Memory models and their implications for concurrency
 - Multi-threading, locking, and the limitations of lock-based concurrency
- Transactions
- Message passing
- Race Conditions

Implications of Multicore

- Memory Models
- Concurrency, threads, and locks

Memory Models and Multicore Systems

- Hardware trends: multicore with non-uniform memory access
 - Increasing numbers of cores, for performance
 - Cache coherency increasingly expensive → cores communicate by message passing, memory is remote
- When do writes made by one core become visible to other cores?
 - What is the **memory model** for the language?
 - Prohibitively expensive for all threads on all cores to have the exact same view of memory (“sequential consistency”)
 - For performance, allow cores inconsistent views of memory, except at synchronisation points; introduce synchronisation primitives with well-defined semantics
 - Hardware guarantees vary between processors
 - Differences hidden by language runtime, provided language has a clear memory model

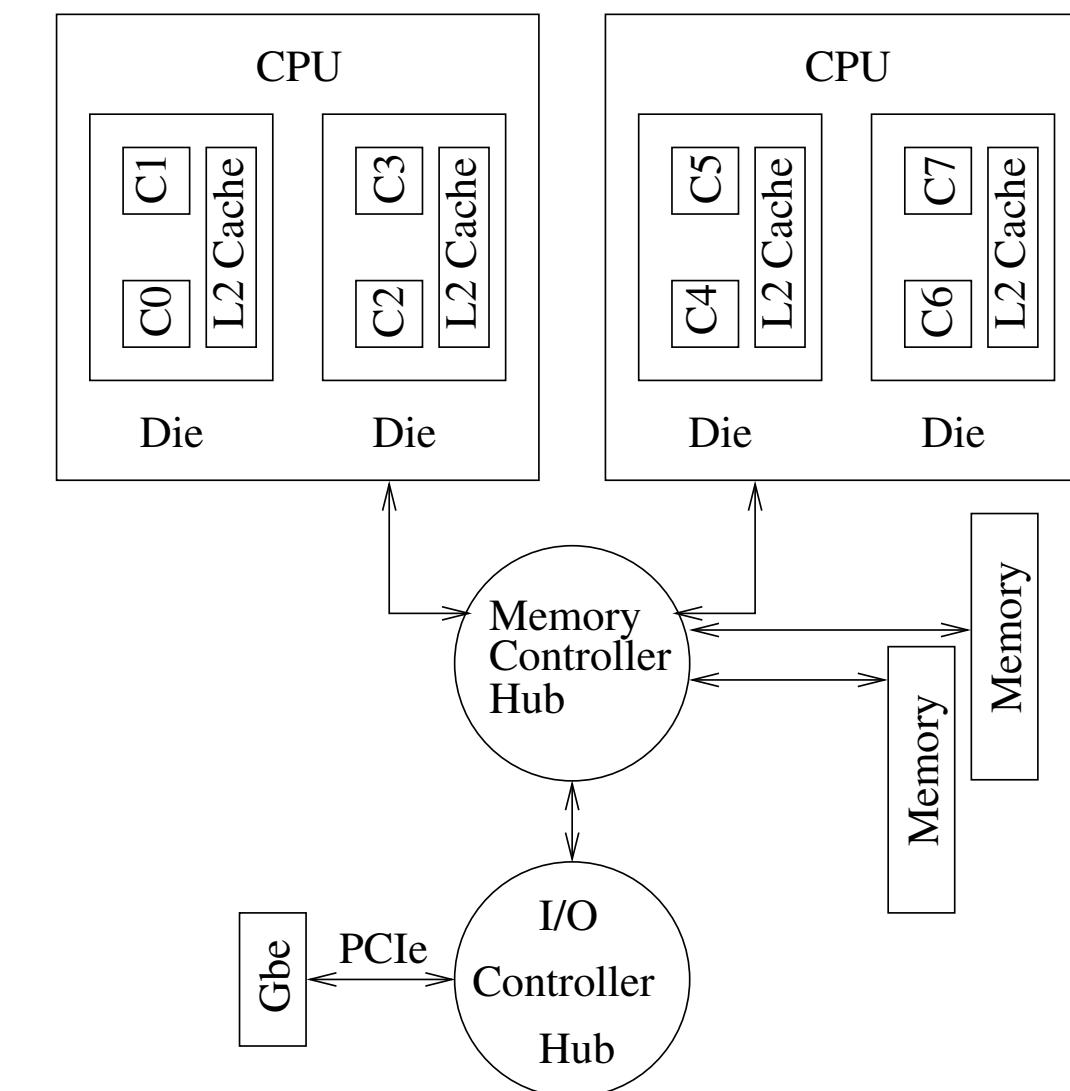


Figure 1. Structure of the Intel system

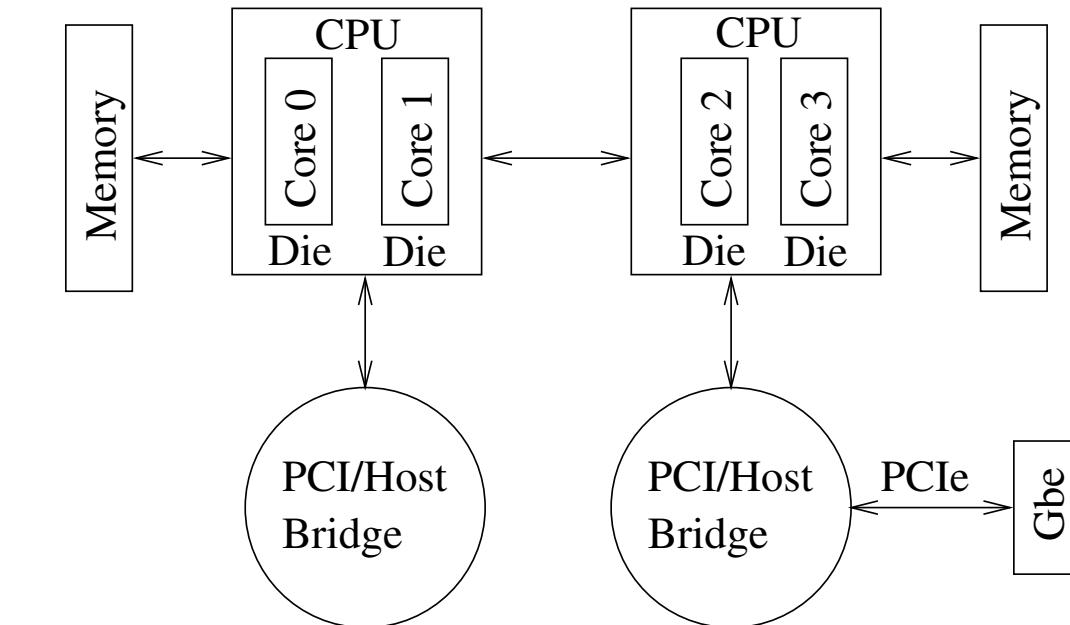


Figure 2. Structure of the AMD system

A. Schrijbach, et al., Embracing diversity in the Barrelfish manycore operating system.
Proc. Workshop on Managed Many-Core Systems, Boston, MA, USA, June 2008. ACM.

Memory Models: Java

- Java has a formally defined memory model
 - Changes to a field are seen in program order *within a thread*
 - Changes to a field made by one thread are visible to other threads as follows:
 - If a **volatile** field is changed, that change is done atomically and immediately becomes visible to other threads
 - If a non-**volatile** field is changed while holding a lock, and that lock is then released by the writing thread and acquired by the reading thread, then the change becomes visible to the reading thread
 - If a new thread is created, it sees the state of the system as if it had just acquired a lock that had just been released by the creating thread
 - If a thread terminates, changes it made become visible to other threads
 - Access to all 32-bit fields is atomic
 - i.e., you can never observe a half-way completed write, even if incorrectly synchronised
 - This is not true for for **long** and **double** fields, which are 64-bits in size, where writes are only atomic if field is **volatile** or if a lock is held

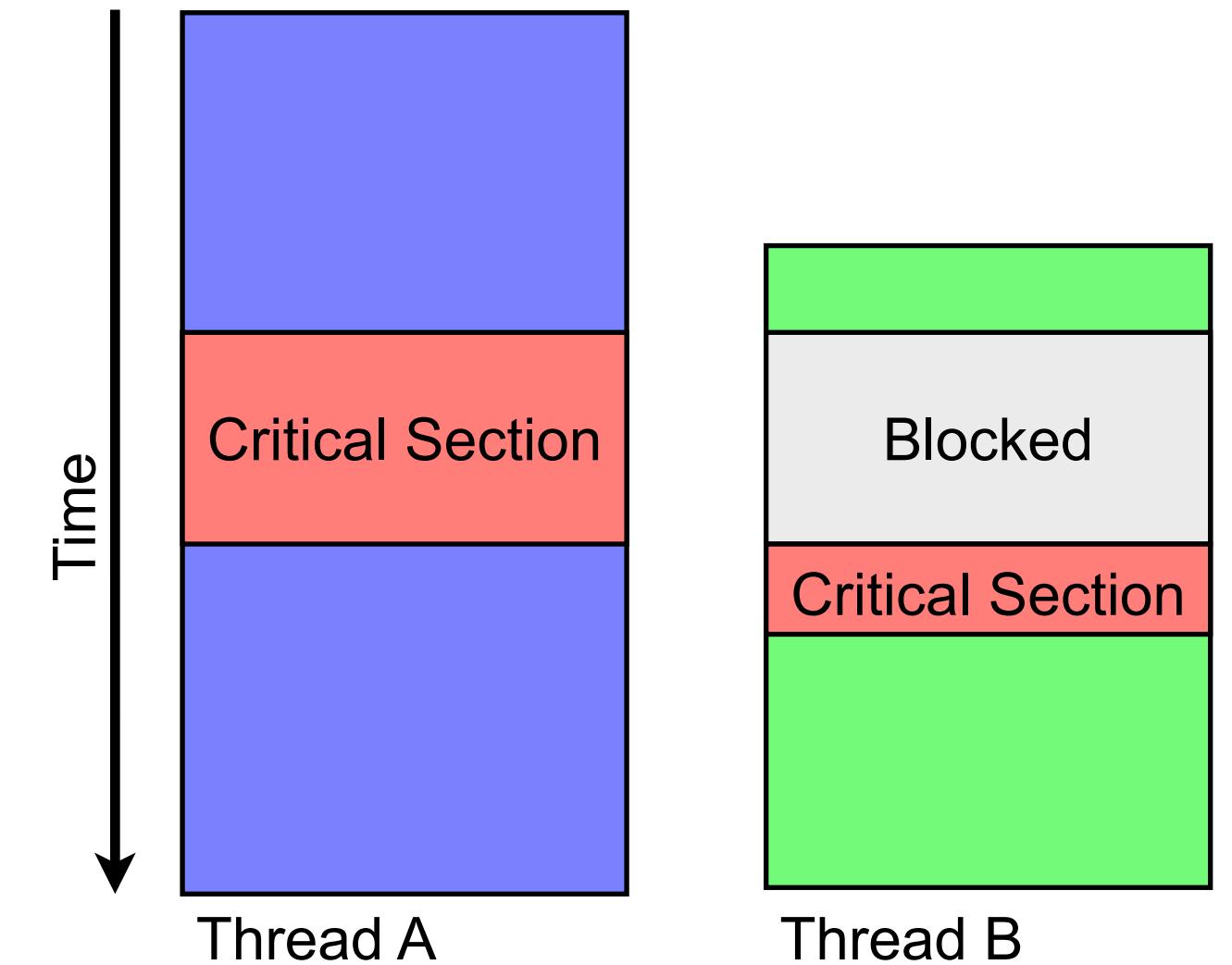
Java Language Specification, Chapter 17
<http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf>

Memory Models: Others

- Java is unusual in having such a clearly-specified memory model
 - Other languages are less well specified, running the risk that new processor designs can subtly break previously working programs
 - C and C++ have historically had *very* poorly specified memory models
 - Latest versions of standards address this, with memory models heavily influenced by the Java memory model
 - Not yet widely implemented
 - Rust does not (yet) have a fully specified memory model
 - Recognised as a limitation – research efforts underway to fix this
 - Complicated by multiplicity of reference types and **unsafe** code

Concurrency, Threads, and Locks

- Operating system exposes concurrency as **processes** and **threads**
 - Processes are isolated with separate memory areas
 - Threads share access to a common pool of memory
 - Most operating systems started with processes and message passing, and added threads later due to programmer demand
- The memory model specifies how concurrent access to shared memory works
 - Synchronisation by explicit locks around critical sections
 - **synchronized** methods and statements in Java
 - **pthread_mutex_lock()**/**pthread_mutex_unlock()** in C
 - Synchronisation by **volatile** fields
 - Limited guarantees about unlocked concurrent access to shared memory

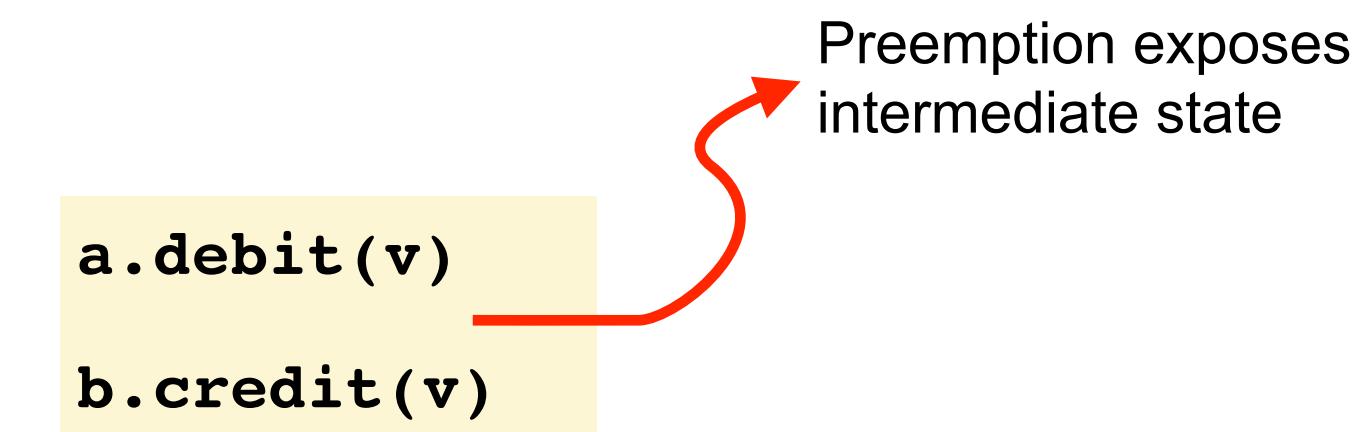


Limitations of Lock-based Concurrency

- Major problems with lock-based concurrency:
 - Difficult to define a memory model that enables good performance, while allowing programmers to reason about the code
 - Difficult to ensure correctness when composing code
 - Difficult to enforce correct locking
 - Difficult to guarantee freedom from deadlocks
 - Failures are silent – errors tend to manifest only under heavy load
 - Balancing performance and correctness difficult – easy to over- or under-lock systems

Composition of Lock-based Code

- Correctness of small-scale code using locks can, in theory, be ensured by careful coding
- A more fundamental issue: lock-based code does not compose to larger scale
 - Assume a correctly locked bank account class, with methods to credit and debit money from an account
 - Want to take money from **a** and move it to **b**, without exposing an intermediate state where the money is in neither account
 - Can't be done without locking all other access to **a** and **b** while the transfer is in progress
 - The individual operations are correct, but the combined operation is not
- This is lack of abstraction a limitation of the lock-based concurrency model, and cannot be fixed by careful coding
- Locking requirements form part of the API of an object



Alternative Concurrency Models

- Concurrency increasingly important
 - Multicore systems now ubiquitous
 - Asynchronous interactions between software and hardware devices
 - Threads and synchronisation primitives problematic
- Are there alternatives that avoid these issues?
 - Transactions
 - Message passing

Implications of Multicore

- Memory Models
- Concurrency, threads, and locks

Managing Concurrency Using Transactions

- Programming model
- Integration into Haskell
- Integration into other languages
- Discussion

Transactions for Managing Concurrency

- An alternative to locking: use *atomic transactions* to manage concurrency
 - A program is a sequence of concurrent atomic actions
 - Atomic actions succeed or fail in their entirety, and intermediate states are not visible to other threads
 - The runtime must ensure actions have the usual ACID properties:
 - **Atomicity** – all changes to the data are performed, or none are
 - **Consistency** – data is in a consistent state when a transaction starts, and when it ends
 - **Isolation** – intermediate states of a transaction are invisible to other transactions
 - **Durability** – once committed, results of a transaction persist
 - Advantages:
 - Transactions can be composed arbitrarily, without affecting correctness
 - Avoid deadlock due to incorrect locking, since there are no locks

```
atomic {  
    a1.debit(v)  
    a2.credit(v)  
}
```

Programming Model

- Simple programming model:
 - Blocks of code can be labelled **atomic** {...}
 - Run concurrently and atomically with respect to every other **atomic** {...} blocks – controls concurrency and ensures consistent data structures
- Implemented via optimistic transactions
 - A thread-local transaction log is maintained, records every memory read and write made by the atomic block
 - When an atomic block completes, the log is *validated* to check that it has seen a consistent view of memory
 - If validation succeeds, the transaction *commits* its changes to memory; if not, the transaction is rolled-back and retried from scratch
 - Progress may be slow if *conflicting* transactions cause repeated validation failures, but will eventually occur

Programming Model – Consequences

- Transactions may be re-run automatically, if their transaction log fails to validate
- Places restrictions on transaction behaviour:
 - Transactions must be referentially transparent
 - Transactions must do nothing irrevocable:
 - Might launch the missiles multiple times, if it gets re-run due to validation failure caused by **doMoreStuff()**
 - Might accidentally launch the missiles if a concurrent transaction modifies **n** or **k** while the transaction is running (will cause transaction failure, but too late to stop the launch)
 - These restrictions must be enforced, else we trade hard-to-find locking bugs for hard-to-find transaction bugs

```
atomic(n, k) {  
    doSomeStuff()  
    if (n > k) then launchMissiles();  
    doMoreStuff();  
}
```

Controlling I/O

- Unrestricted I/O breaks transaction isolation
 - Reading and writing files
 - Sending and receiving data over the networks
 - Taking mouse or keyboard input; changing the display
- Require language control of when I/O is performed
 - Remove global functions to perform I/O from the standard library
 - Add an **I/O context** object, local to `main()`, passed explicitly to functions that need to perform I/O
 - Compare sockets, that behave in this manner, with file I/O that typically does not
 - I/O functions (e.g., `printf()` and friends) then become methods on the I/O context object
 - The I/O context is not passed to transactions, so they cannot perform I/O
 - Example: the IO monad in Haskell

Controlling Side Effects

- Code that has side effects must be controlled
 - Pure and referentially transparent functions can be executed normally
 - Functions that only perform memory actions can be executed normally, *provided* transaction log tracks the memory actions and validates them before the transaction commits – and can potentially roll them back
 - A *memory action* is an operation that manipulates data on the heap, that could be seen by other threads
 - Tracking memory actions can be done by language runtime (STM; software transactional memory), or via hardware enforced transactional memory behaviour and rollback
- Similar principle as controlling I/O
 - Disallow unrestricted heap access – only see data in transaction context
 - Pass transaction context explicitly to transactions; this has operations to perform transactional memory operations, and rollback if the transaction fails to commit
 - Very similar to the state monad in Haskell

Monadic STM Implementation (1)

- Monads → way to control side-effects in functional languages
 - A monad \mathbf{M} \mathbf{a} describes an action (i.e., a function) that, produces a result of type \mathbf{a} , that can be performed in the context \mathbf{M}
 - Along with rules for chaining operations in that context
- A common use is for controlling I/O operations:
 - The **putChar** function takes a character, operates on the **IO** context to add the character, and returns nothing
 - The **getChar** operates on the **IO** context to return a character
 - The main function has an IO context, that wraps and performs other actions
 - The definition of the I/O monad type ensures that a function that is not passed the IO context cannot perform I/O operations
 - One part of a software transactional memory implementation: ensure type of the **atomic** `{...}` function does not allow it to be passed an IO context, hence preventing I/O

```
putChar :: Char -> IO ()  
getChar :: IO Char
```

Monadic STM Implementation (2)

- How to track side-effecting memory actions?
 - Define an STM monad to wrap transactions
 - Based on the state monad; manages side-effects via a **TVar** type
 - The **newTVar** function takes a value of type a, returns new **TVar** to hold the value, wrapped in an STM monad
 - **readTVar** takes a **TVar** and returns an STM context; when performed this returns the value of that **TVar**; **writeTVar** function takes a **TVar** and a value, returns an STM context that can validate the transaction and commit the value to the **TVar**
 - The **atomic** {...} function operates in an STM context and returns an IO context that performs the operations needed to validate and commit the transaction
 - The **newTVar**, **readTVar**, and **writeTVar** functions need an STM action, and so can only run in the context of an atomic block that provides such an action
 - I/O prohibited within transactions, since operations in **atomic** {...} don't have access to I/O context

```
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

```
atomic :: STM a -> IO a
```

Integration into Haskell

- Transactional memory is a good fit with Haskell
 - Pure functions and monads ensure transaction semantics are preserved
 - Side-effects are contained in **STM** context of an **atomic** `{...}` block
 - The **TVar** implementation is responsible for tracking side effects
 - The **atomic** `{...}` block validates, then commits the transaction (by returning an action to perform in the IO context)
 - A **TVar** requires an **STM** context, but these are only available in an **atomic** `{...}` block; can't update a **TVar** outside a transaction, so can't break atomicity guidelines – Haskell doesn't allow unrestricted heap access via pointers, so can't subvert
 - I/O cannot be performed within an **atomic** `{...}` block
 - The transaction is not in the IO context

Integration into Other Languages

- Atomic transactions Haskell are very powerful – but rely on the type system to ensure safe composition and retry
- Integration into mainstream languages is difficult
 - Most languages cannot enforce use of pure functions
 - Most languages cannot limit the use of I/O and side effects
 - Transaction memory can be used without these, but requires programmer discipline to ensure correctness – and has silent failure modes
- Unclear if the transactional approach generalises to other languages

Further Reading

- Is transactional memory a realistic technique?
- Do its requirements for a purely functional language, with controlled I/O, restrict it to being a research toy?
- How much benefit can be gained from transactional memory in more traditional languages?

DOI:10.1145/1378704.1378725

Composable Memory Transactions

By Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy

Abstract
Writing concurrent programs is notoriously difficult and is of increasing practical importance. A particular source of concern is that even correctly implemented concurrency abstractions cannot be composed together to form larger abstractions. In this paper we present a concurrency model, based on *transactional memory*, that offers far richer composition. All the usual benefits of transactional memory are present (e.g., freedom from low-level deadlock), but in addition we describe modular forms of *blocking* and *choice* that were inaccessible in earlier work.

1. INTRODUCTION
The free lunch is over.²⁵ We have been used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If we want our programs to run faster, we must learn to write parallel programs.

Writing parallel programs is notoriously tricky. Mainstream lock-based abstractions are difficult to use and they make it hard to design computer systems that are reliable and scalable. Furthermore, systems built using locks are difficult to compose without knowing about their internals.

To address some of these difficulties, several researchers (including ourselves) have proposed building programming language features over *software transactional memory* (STM), which can perform groups of memory operations atomically.²³ Using transactional memory instead of locks brings well-known advantages: freedom from deadlock and priority inversion, automatic roll-back on exceptions or timeouts, and freedom from the tension between lock granularity and concurrency.

Early work on software transactional memory suffered several shortcomings. Firstly, it did not prevent transactional code from bypassing the STM interface and accessing data directly at the same time as it is being accessed within a transaction. Such conflicts can go undetected and prevent transactions executing atomically. Furthermore, early STM systems did not provide a convincing story for building operations that may block—for example, a shared work-queue supporting operations that wait if the queue becomes empty.

Our work on STM-Haskell set out to address these problems. In particular, our original paper makes the following contributions:

- We re-express the ideas of transactional memory in the setting of the purely functional language Haskell (Section 3). As we show, STM can be expressed particularly elegantly in a declarative language, and we are able to use Haskell's type system to give far stronger guarantees than are conventionally possible. In particular, we guarantee “strong atomicity”¹⁵ in which transactions always appear to execute atomically, no matter what the rest of the program is doing. Furthermore transactions are compositional: small transactions can be glued together to form larger transactions.
- We present a modular form of blocking (Section 3.2). The idea is simple: a transaction calls a *retry* operation to signal that it is not yet ready to run (e.g., it is trying to take data from an empty queue). The programmer does not have to identify the condition which will enable it; this is detected automatically by the STM.
- The *retry* function allows possibly blocking transactions to be composed in *sequence*. Beyond this, we also provide *orElse*, which allows them to be composed as *alternatives*, so that the second is run if the first retries (see Section 3.4). This ability allows threads to wait for many things at once, like the Unix *select* system call—except that *orElse* composes, whereas *select* does not.

Everything we describe is fully implemented in the Glasgow Haskell Compiler (GHC), a fully fledged optimizing compiler for Concurrent Haskell; the STM enhancements were incorporated in the GHC 6.4 release in 2005. Further examples and a programmer-oriented tutorial are also available.¹⁹

Our main war cry is *compositionality*: a programmer can control atomicity and blocking behavior in a modular way that respects abstraction barriers. In contrast, lock-based approaches lead to a direct conflict between abstraction and concurrency (see Section 2). Taken together, these ideas offer a qualitative improvement in language support for modular concurrency, similar to the improvement in moving from assembly code to a high-level language. Just as with assembly code, a programmer with sufficient time and skills may obtain better performance programming directly with low-level concurrency control mechanisms rather than transactions—but for all but the most demanding applications, our higher-level STM abstractions perform quite well enough.

This paper is an abbreviated and polished version of an earlier paper with the same title.⁹ Since then there has been a tremendous amount of activity on various aspects of transactional memory, but almost all of it deals with the question of *atomic memory update*, while much less attention is paid to our central concerns of *blocking* and *synchronization* between threads, exemplified by *retry* and *orElse*. In our view this is a serious omission: locks without condition variables would be of limited use.

Transactional memory has tricky semantics, and the original paper gives a precise, formal semantics for transactions, as well as a description of our implementation. Both are omitted here due to space limitations.

AUGUST 2008 | VOL. 51 | NO. 8 | COMMUNICATIONS OF THE ACM | 91

T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy,
“Composable Memory Transactions”, Communications of the
ACM, 51(8), August 2008. DOI:10.1145/1378704.1378725.

<http://www.cmi.ac.in/~madhavan/courses/pl2009/reading-material/harris-et-al-cacm-2008.pdf>

Managing Concurrency Using Transactions

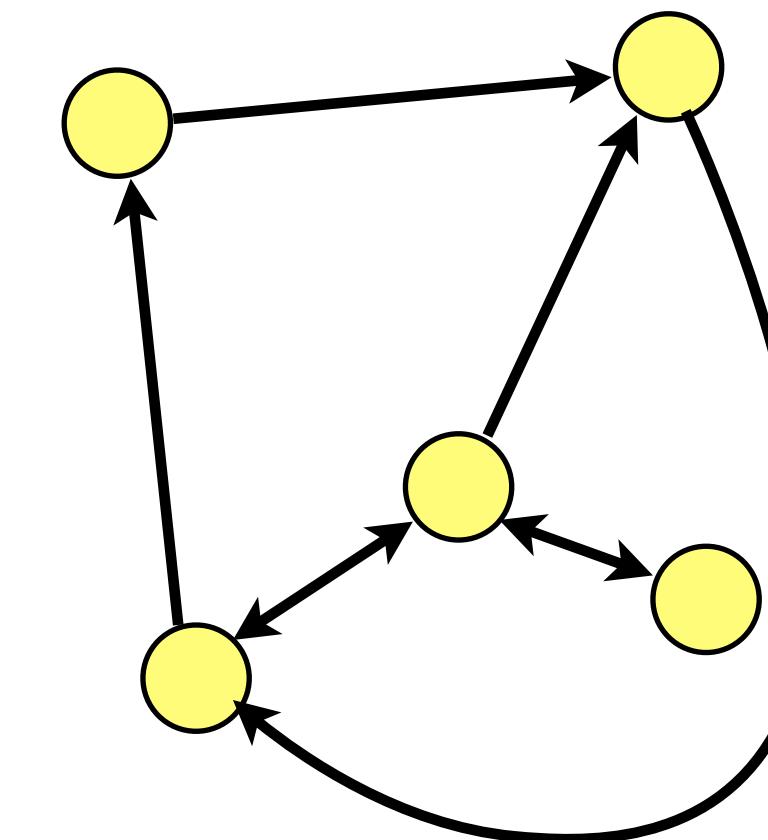
- Programming model
- Integration into Haskell
- Integration into other languages
- Discussion

Message Passing Systems

- Actors and message passing
- Structure of communication
- Implementation in Scala and Rust

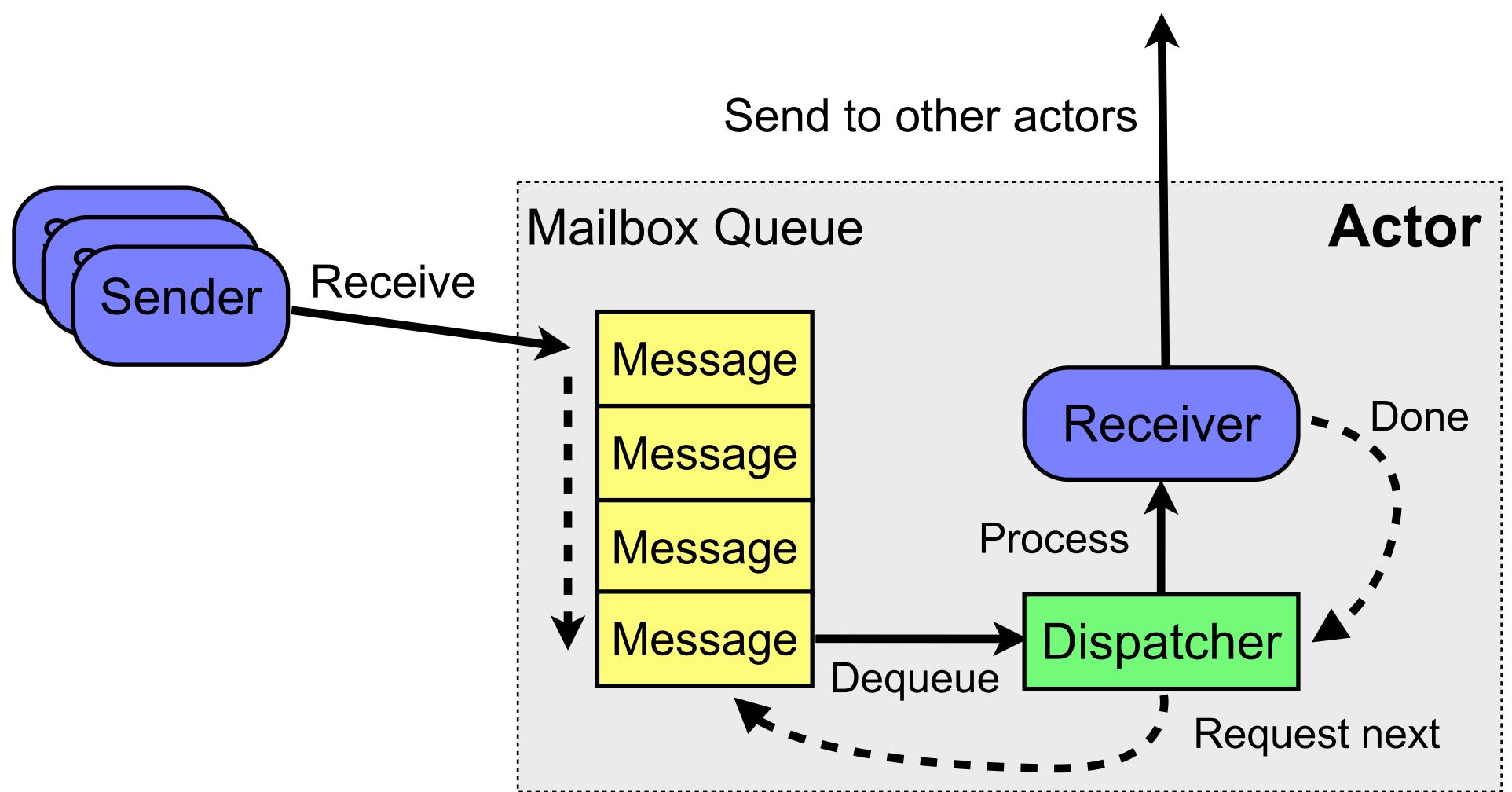
Message Passing Systems

- System is structured as a set of communicating processes, *actors*, with no shared mutable state
- All communication via exchange of messages
 - Messages are generally required to be immutable – data conceptually copied between processes
 - Some systems use linear types to ensure messages are not referenced after they are sent, allowing mutable data to be safely transferred
- Implementation
 - Implementation within a single system usually built with shared memory and locks, passing a reference to the message – rely on correct locking of message passing implementation
 - Trivial to distribute, by sending the message down a network channel – the runtime needs to know about the network, but the application can be unaware that the system is distributed



Message Handling

- Receivers pattern match against messages
 - Match against message types, not just values
 - Type system can ensure an exhaustive match
- Messages queued for processing
 - Dispatcher manages a thread pool servicing receiver components of the actors
 - Receivers operate in message processing loop – single-threaded, with no concern for concurrency
 - Sent messages enqueued for processing by other actors



Types of Message Passing

- Several different message passing system designs:
 - Synchronous vs asynchronous
 - Statically or dynamically typed
 - Direct or indirect message delivery
- Each has advantages and disadvantages

Types of Message Passing: Interaction Models

- Message passing can involve rendezvous between sender and receiver
 - A synchronous message passing model – sender waits for receiver
 - Alternatively, communication may be asynchronous
 - The sender continues immediately after sending a message
 - Message is buffered, for later delivery to the receiver
 - Synchronous rendezvous can be simulated by waiting for a reply

Types of Message Passing: Typed Communication

- Statically-typed communication
 - Explicitly define the types of message that can be transferred
 - Compiler checks that receiver can handle all messages it can receive – robustness, since a receiver is guaranteed to understand all messages
- Dynamically-typed communication
 - Communication medium conveys any type of message; receiver uses pattern matching on the received message types to determine if it can respond to the messages
 - Potentially leads to run-time errors if a receiver gets a message that it doesn't understand

Types of Message Passing: Naming

- Are messages sent between named processes or indirectly via channels?
 - Some systems directly send *messages* to actors (processes), each of which has its own mailbox
 - Others use explicit *channels*, with messages being sent indirectly to a mailbox via a channel
- Explicit channels require more plumbing, but the extra level of indirection between sender and receiver may be useful for evolving systems
- Explicit channels are a natural place to define a communications protocol for statically typed messages

Implementations

- Message passing starting to see wide deployment, with two widely used architectures:
 - Dynamically typed with direct delivery
 - Erlang programming language (<https://www.erlang.org/>)
 - Scala programming language (<http://www.scala-lang.org>) and Akka library (<http://akka.io>)
 - Dynamically typed – any type of message may be sent to any receiver
 - Messages sent directly to named actors, not via channels
 - Both provide transparent distribution of processes in a networked system
 - Statically typed, with explicit channels
 - Rust programming language (<https://www.rust-lang.org>)
 - Use asynchronous statically typed messages passed via explicit channels

Example: Scala+Akka

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props

class HelloActor extends Actor {
  def receive = {
    case "hello" => println("hello back at you")
    case _          => println("huh?")
  }
}

object Main extends App {
  // Initialise actor runtime
  val runtime = ActorSystem("HelloSystem")

  // Create an actor, running concurrently
  val helloActor = runtime.actorOf(Props[HelloActor], name = "helloactor")

  // Send it some messages
  helloActor ! "hello"
  helloActor ! "buenos dias"
}
```

The actor comprises a receive loop that reacts to messages as they're received

Complete program is a collection of actors that exchange messages

Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
    let (tx, rx) = channel();

    thread::spawn(move || {
        let _ = tx.send(42);
    });

    match rx.recv() {
        Ok(value) => {
            println!("Got {}", value);
        }
        Err(error) => {
            // An error occurred...
        }
    }
}
```

Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
    let (tx, rx) = channel();

    thread::spawn(move || {
        let _ = tx.send(42);
    });

    match rx.recv() {
        Ok(value) => {
            println!("Got {}", value);
        }
        Err(error) => {
            // An error occurred...
        }
    }
}
```



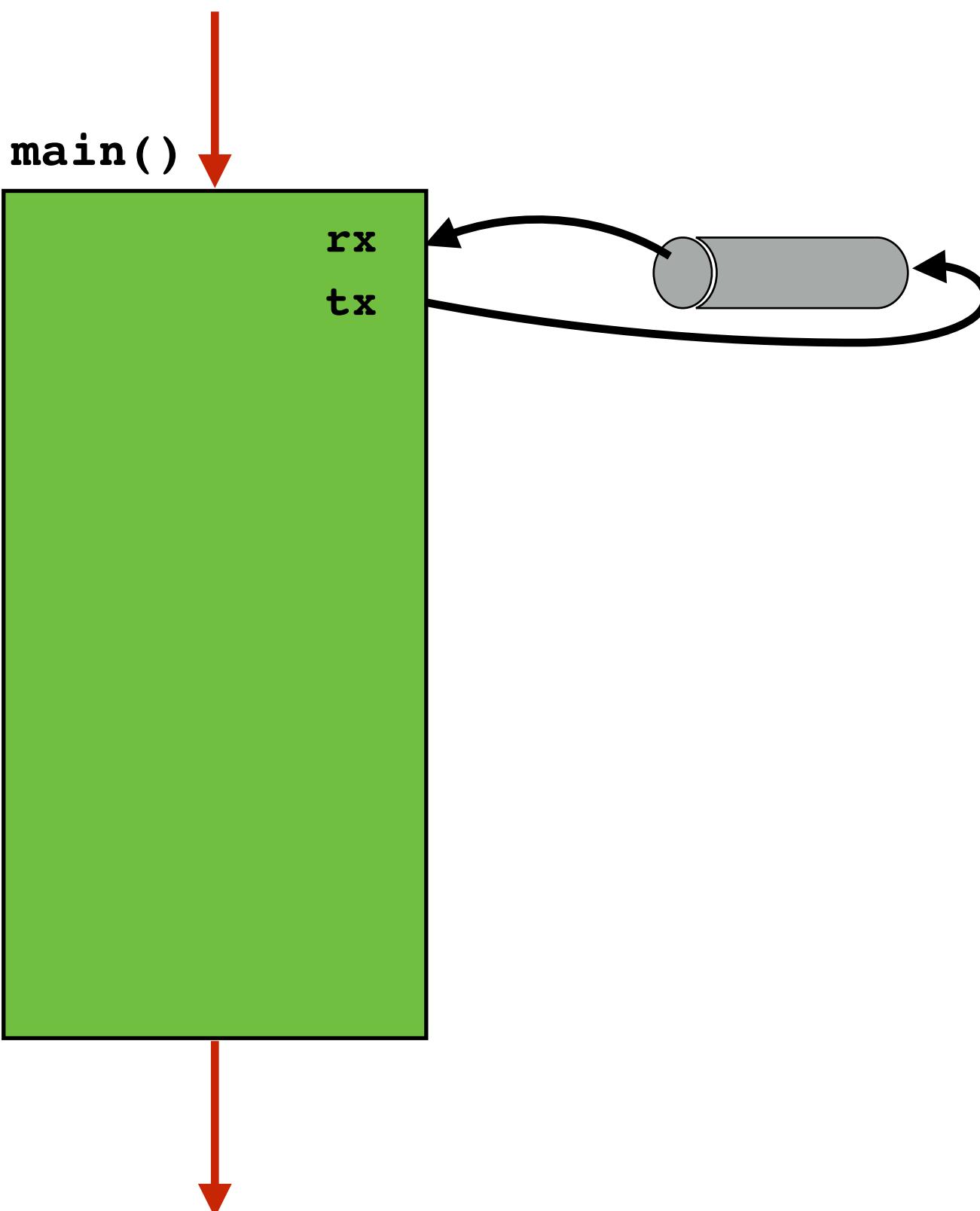
Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
    let (tx, rx) = channel();

    thread::spawn(move || {
        let _ = tx.send(42);
    });

    match rx.recv() {
        Ok(value) => {
            println!("Got {}", value);
        }
        Err(error) => {
            // An error occurred...
        }
    }
}
```



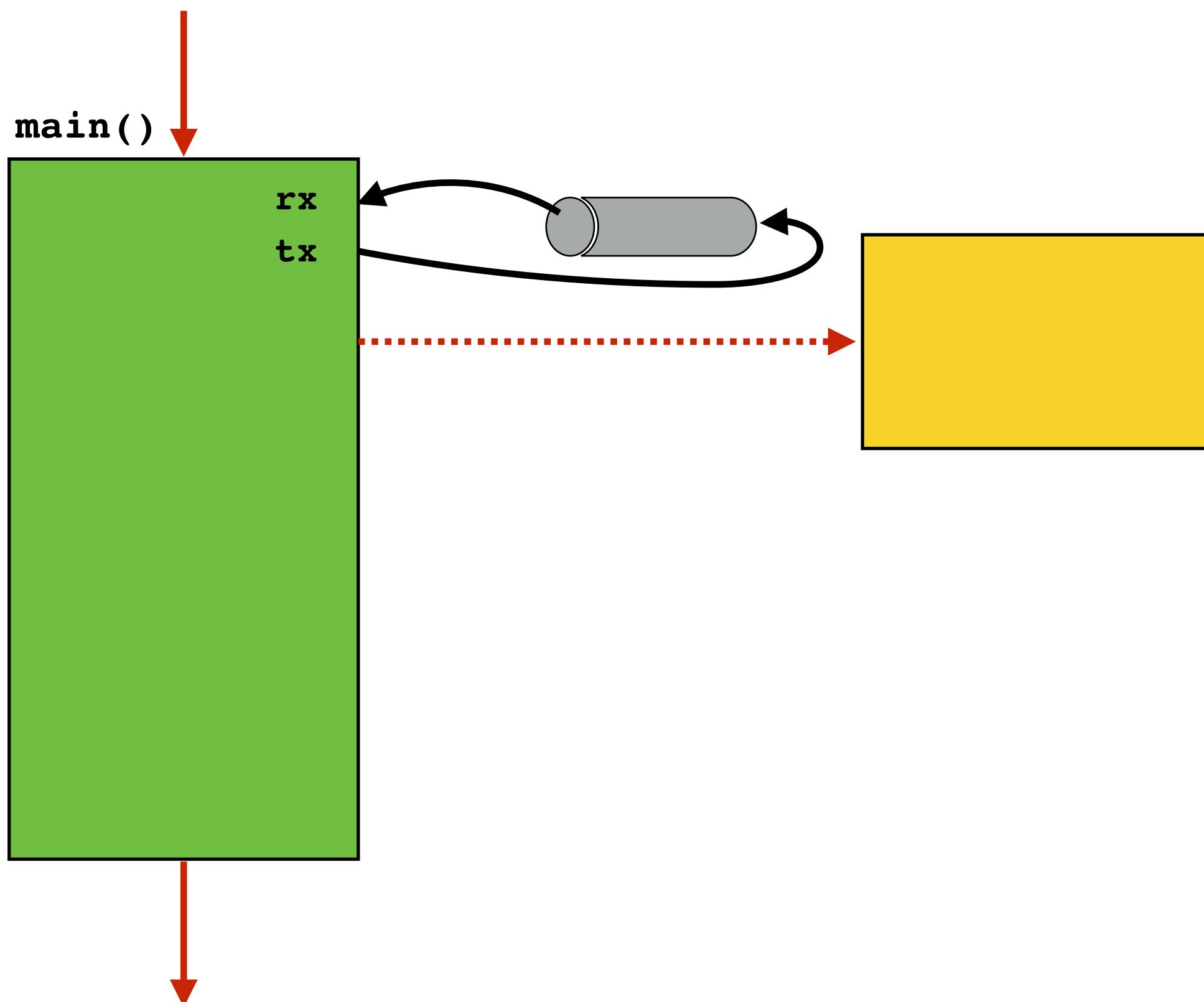
Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
    let (tx, rx) = channel();

    thread::spawn(move|| {
        let _ = tx.send(42);
    });

    match rx.recv() {
        Ok(value) => {
            println!("Got {}", value);
        }
        Err(error) => {
            // An error occurred...
        }
    }
}
```



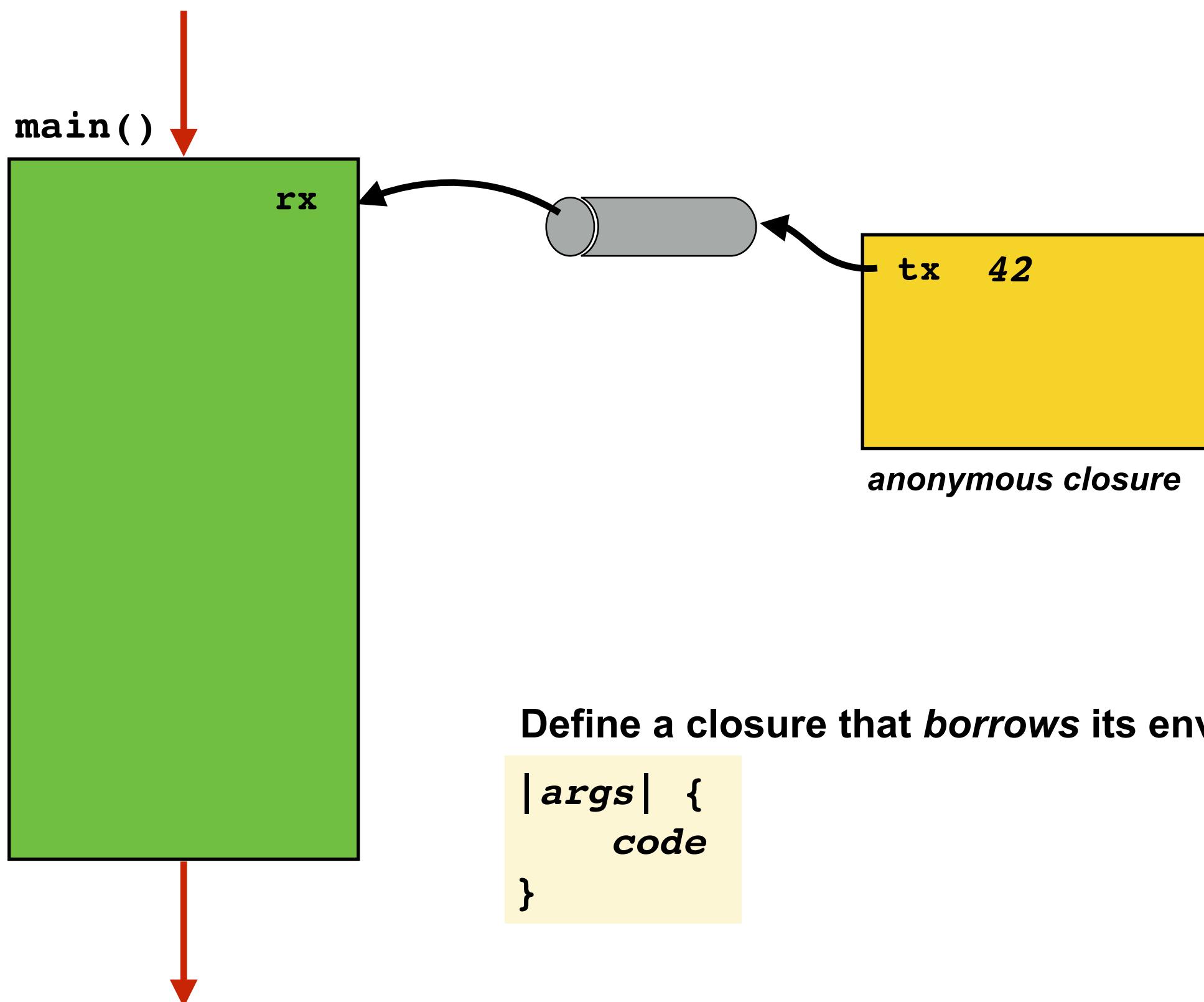
Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
    let (tx, rx) = channel();

    thread::spawn(move || {
        let _ = tx.send(42);
    });

    match rx.recv() {
        Ok(value) => {
            println!("Got {}", value);
        }
        Err(error) => {
            // An error occurred...
        }
    }
}
```



Define a closure that *borrow*s its environment:

```
|args| {  
    code  
}
```

Define a closure that *takes ownership* of its environment:

```
move |args| {  
    code  
}
```

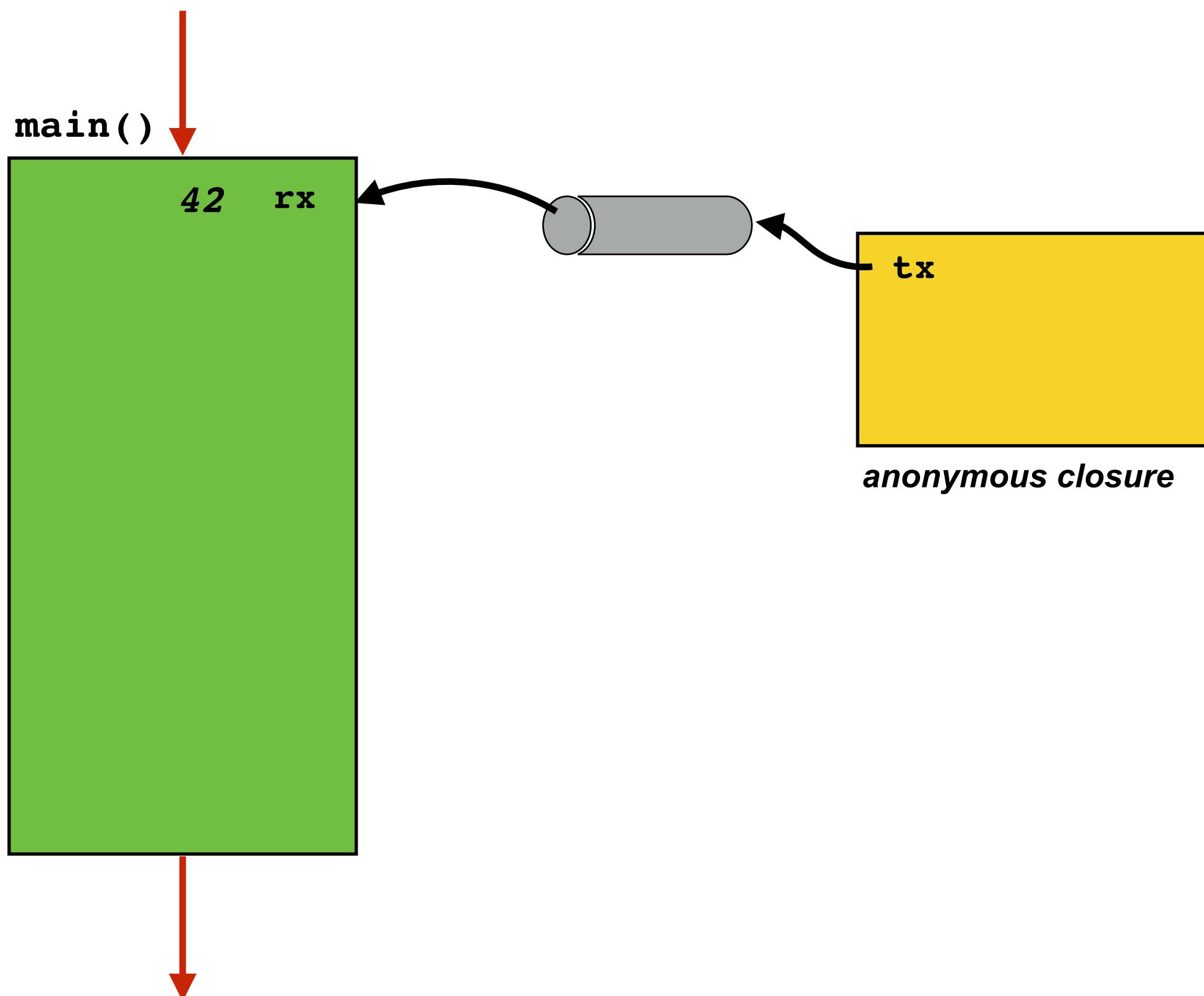
Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
    let (tx, rx) = channel();

    thread::spawn(move || {
        let _ = tx.send(42);
    });

    match rx.recv() {
        Ok(value) => {
            println!("Got {}", value);
        }
        Err(error) => {
            // An error occurred...
        }
    }
}
```



Trade-offs

- The two approaches behave quite differently:
 - **Scala+Akka** let weakly coupled processes to communicate via asynchronous and dynamically typed messages:
 - Expressive, flexible, and extensible actor model
 - Robust framework for error handling via separate processes
 - Relative ease of upgrading running systems via dynamic actor insertion
 - Checking happens at run time, so guarantees of robustness are probabilistic
 - **Rust** uses statically typed message passing provides compile-time checking that a process can respond to messages
 - But, requires more plumbing to connect channels
 - Has more explicit error handling
 - The usual static vs. dynamic typing debate

Message Passing Systems

- Actors and message passing
- Immutable data
- Ownership and race conditions

Race Conditions

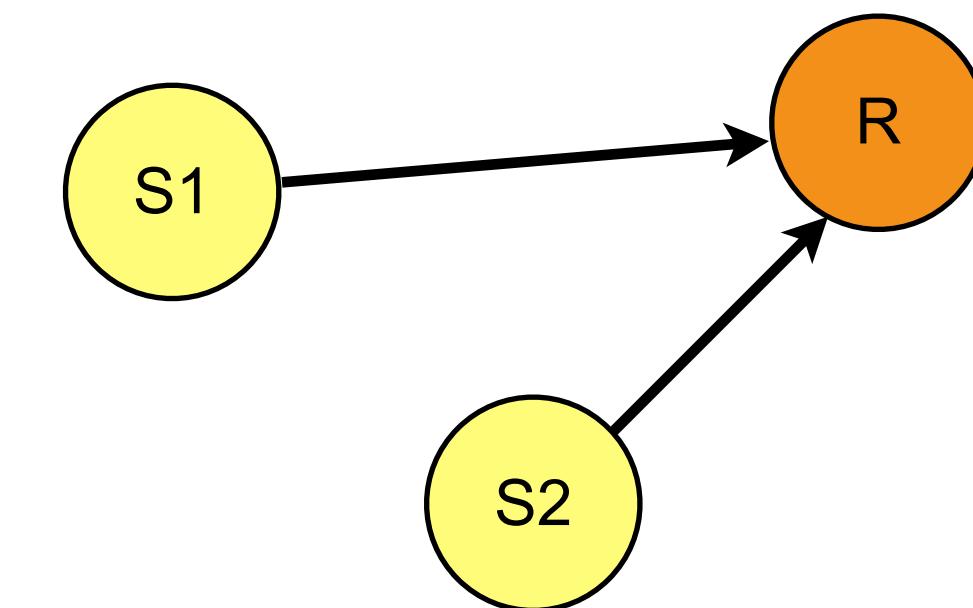
- What is a race condition?
- Race conditions in message passing and shared memory systems

What is a Race Condition?

- A race condition can occur when the behaviour of a system depends on the relative timing of different actions – or when a shared value is modified without coordination
- Introduces non-deterministic behaviour and hard-to-debug problems
 - Difficult to predict exact timing of program behaviour
 - Difficult to predict effects of asynchronous, uncontrolled, modification to shared values

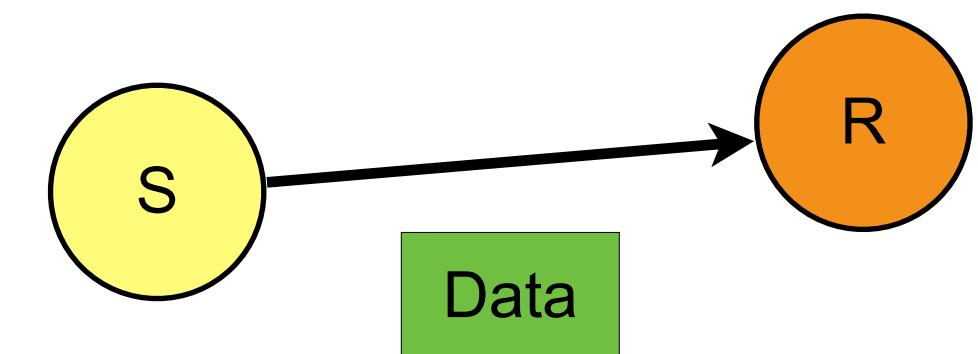
Race Conditions: Message Races

- In message passing systems, messages can be received from multiple senders
- Runtime ensures receiver processes messages sequentially, in the order they are received, but order of receipt can vary due to system and network load, external events, etc.
 - A **race condition** occurs when messages arrive in unpredictable order
 - A **deadlock** occurs when a cycle forms, with actors waiting for messages from each other
- Structure communication patterns to avoid



Race Conditions: Data Races

- In shared memory systems, data is conceptually moved from sender to receiver
 - In practice, a **reference** to the data is often copied, for performance reasons, and the underlying data remains in place
 - A **data race** condition occurs if the data is modified after it is sent, and the modification is visible to the receiver via the reference
 - Unpredictable if the receiver sees the old or new version, depending on timing of changes, scheduling, etc.
 - Two approaches to avoid data races: **immutable data** or **ownership tracking**



Avoiding Data Races: Immutable Data

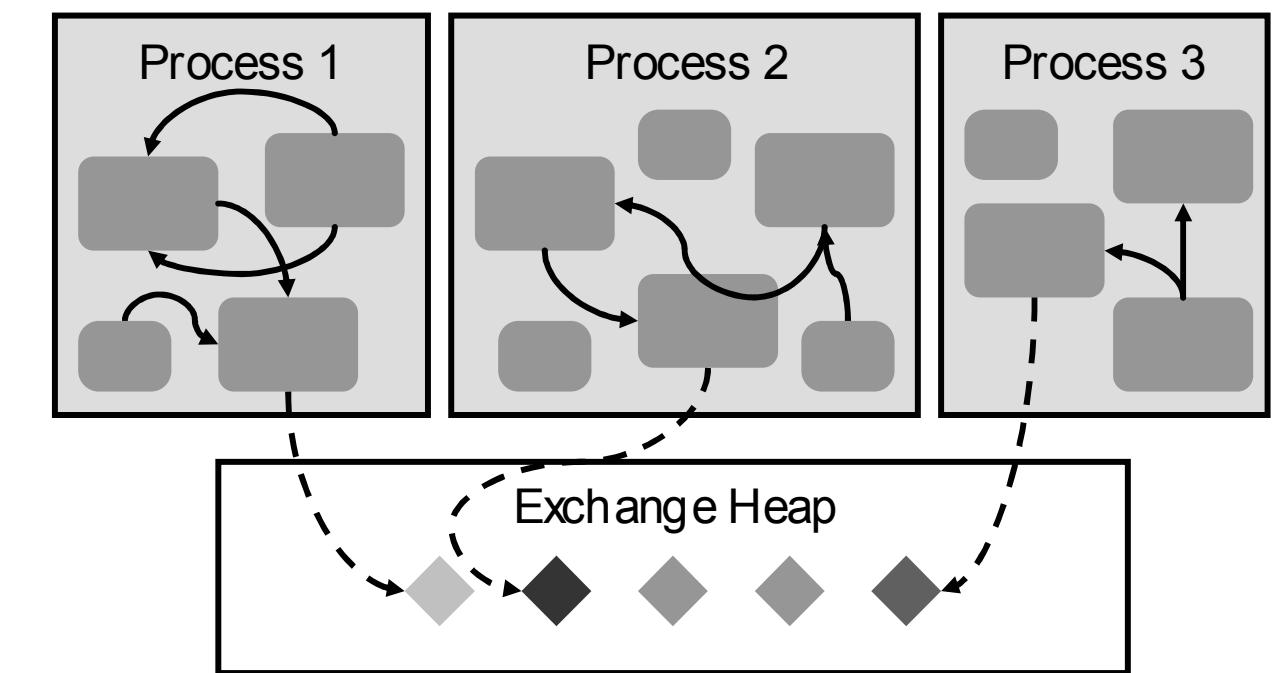
- Race conditions cannot occur if data is not modified
 - Ensure any objects to be sent between threads is immutable
 - Erlang ensures this in the language → *all* variables are immutable
 - Scala+Akka requires programmer discipline → potential race conditions if message data modified after message sent

Avoiding Data Races: Ownership Transfer

- Race conditions cannot occur if data is not shared
 - Ensure ownership of an object is transferred, so the sender cannot access the object after it has been sent
 - Natural fit for Rust:
 - Standard library provides a **channel** abstraction
 - The **send()** function takes ownership of the data to be sent
 - The **recv()** function returns ownership of the received data
 - The usual ownership rules in the type system ensure the data is not accessible once it has been sent
 - If sender can't access object once it's been sent, can't change it → race condition prevented

Aside: Efficiency of Message Passing

- Assuming immutable message or linear types, message passing has efficient implementation
 - Copy message data in distributed systems
 - Pass pointer to data in shared memory systems
 - Neither case needs to consider shared access to message data
- Garbage collected systems often allocate messages from a shared *exchange heap*
 - Collected separately from per-process heaps
 - Expensive to collect, since data in exchange heap owned by multiple threads – need synchronisation
 - Per-process heaps can be collected independently and concurrently – ensures good performance



Source: G. Hunt *et al.*, Sealing OS processes to improve dependability and safety. In Proc. EuroSys 2007, Lisbon, Portugal. DOI 10.1145/1272996.1273032

Summary

- Message passing as an alternative concurrency mechanism
- Increasingly popular
 - Erlang, Scala+Akka (or Java+Akka...)
 - Rust
 - Go, ZeroMQ, etc. – unchecked message passing
- Easy to reason about, simple programming model
 - Provided data is immutable, or ownership is tracked

Summary

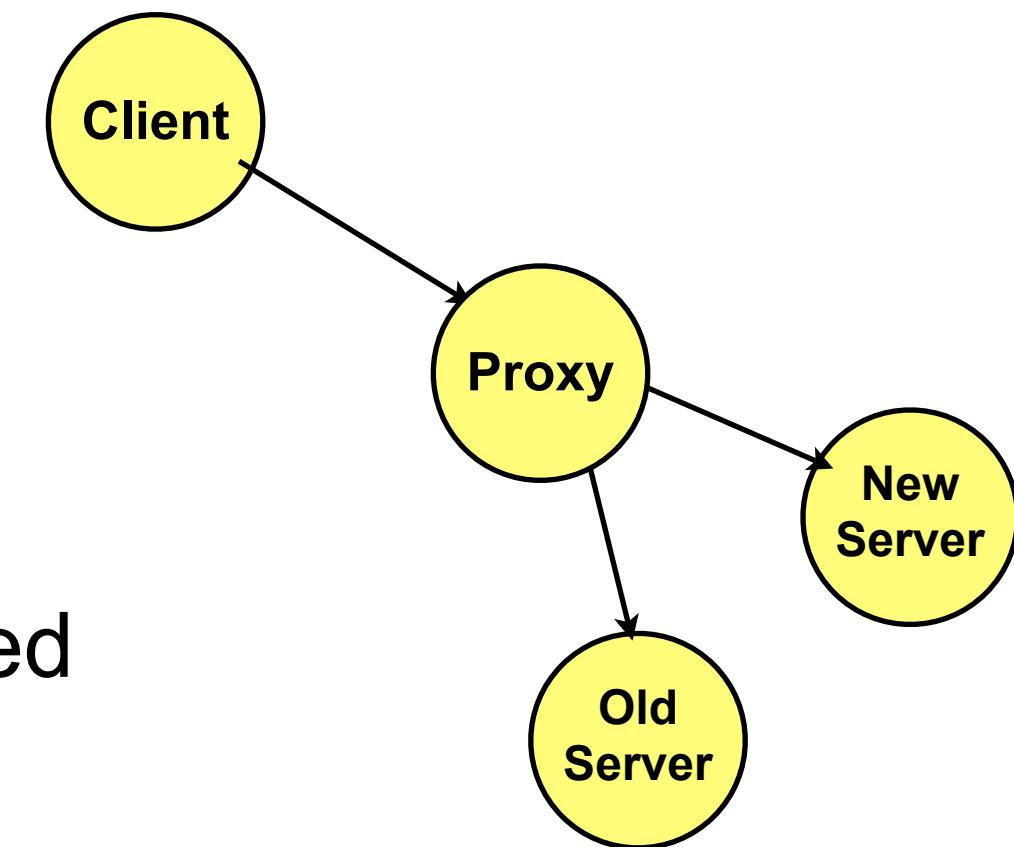
- Concurrency and memory models
- Transactions
- Message Passing

Robustness of Message Passing Systems

- System Upgrade and Evolution
- Error Handling
- Erlang

System Upgrade and Evolution

- Message passing allows for easy system upgrade
 - Rather than pass messages directly to server, pass them via proxy
 - Proxy can load a new version of the server and redirect messages, without disrupting existing clients
 - Eventually, all clients are talking to the new server; old server is garbage collected
 - Allows for gradual transparent system upgrade without disrupting service
- Use of dynamic typing can make the upgrade easier
 - New components of the system can generate additional messages, which are ignored by old components
 - Supervisor hierarchy allows system to notice if components fail, and fallback to known good version
 - Backwards compatible extensions are simple to add in this manner



Error Handling

- The system is massively concurrent – errors in one part can be handled elsewhere
- Error handling philosophy in Erlang:
 - Let some other process do the error recovery
 - If you can't do what you want to do, die
 - Let it crash
 - Do not program defensively
- Be concerned with the overall system reliability, not the reliability of any one component

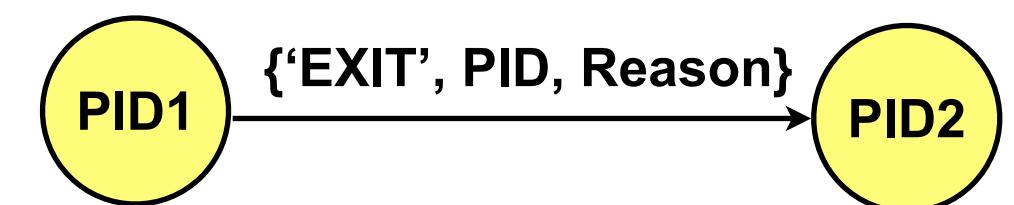
J. Armstrong, “Making reliable distributed systems in the presence of software errors”, PhD thesis, KTH, Stockholm, December 2003,
http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf

Let It Crash

- In a single-process system, that process must be responsible for handling errors
 - If the single process fails, then the entire application has failed
- In a multi-process system, each individual process is less precious – it's just one of many
 - Changes the philosophy of error handling
 - A process which encounters a problem should not try to handle that problem – instead, fail loudly, cleanly, and quickly “let it crash”
 - Let another process cleanup and deal with the problem
- Processes become much simpler, since they’re not cluttered with error handling code

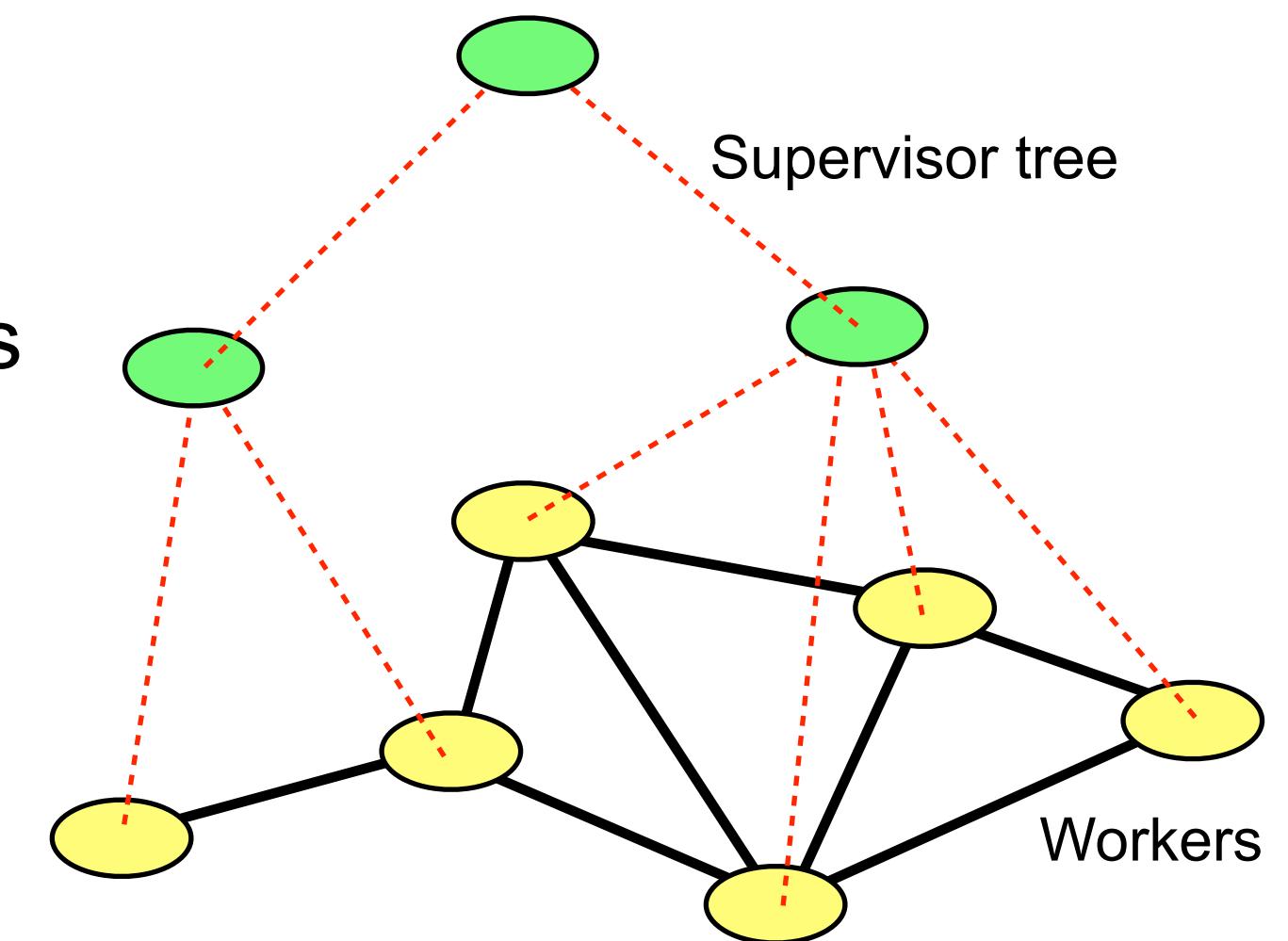
Remote Error Handling

- How to handle errors in a concurrent distributed system?
 - Isolate the problem, let an unaffected process be responsible for recovery
 - Don't trust the faulty component
 - Analogy to hardware fault tolerance
- Processes are linked, runtime is set to trap errors and send a message to the linked process on failure
 - e.g., process PID2 has requested notification of failure of PID1; runtime sends an “EXIT” message on failure, to tell PID2 that PID1 failed, and why
 - Process PID2 then restarts PID1, and any other dependent processes



Supervision Hierarchies

- Organise problems into tree-structured groups of processes, letting the higher nodes in the tree monitor and correct errors in the lower nodes
 - Supervision trees are trees of supervisors – processes that monitor other processes in the system
 - Supervisors monitor workers – which perform tasks – or other supervisors
 - Workers are instances of behaviours – processes whose operation is characterised by callback functions (i.e., the Erlang equivalent of objects)
 - E.g., server, event handler, finite state machine, supervisor, application
- Abstract common behaviours into objects
- Workers managed by supervisor processes that restart them in the case of failure, or otherwise handle errors



Robustness of Erlang Systems

- Example: Ericsson AXD301 ATM switch
 - Dimensioned to handle ~50,000 simultaneous flows with ~120 in setup or teardown phase at any one time; processes traffic at 160Gbps (16 x 10Gbps links); written in Erlang
 - 99.9999999% reliable in real-world deployments on 11 routers at major Ericsson customer (0.5 seconds downtime per year)
 - Failures do occur, but are recovered by supervision hierarchy and distributed error recovery
 - Employs restart-and-recover semantics per-connection so failures disrupt only one connection out of thousands



Images from: S. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, and G. Wicklund, "AXD 301: A new generation ATM switching system", Ericsson Review, 1998.

Discussion

- The let-it-crash philosophy changes error handling, moving it out-of-process
- There are a few compelling case studies to show it can work well in some domains
- Is this a generally appropriate error-handling tool?

Further Reading

- J. Armstrong, “Erlang”, Communications of the ACM, 53(9), September 2010, DOI:10.1145/1810891.1810910
- Does the programming model make sense?
- Does the reliability model (“let it crash”) make sense?

contributed articles

DOI:10.1145/1810891.1810910

The same component isolation that made it effective for large distributed telecom systems makes it effective for multicore CPUs and networked applications.

BY JOE ARMSTRONG

Erlang

ERLANG IS A concurrent programming language designed for programming fault-tolerant distributed systems at Ericsson and has been (since 2000) freely available subject to an open-source license. More recently, we've seen renewed interest in Erlang, as the Erlang way of programming maps naturally to multicore computers. In it the notion of a process is fundamental, with processes created and managed by the Erlang runtime system, not by the underlying operating system. The individual processes, which are programmed in a simple dynamically typed functional programming language, do not share memory and exchange data through message passing, simplifying the programming of multicore computers.

Erlang^a is used for programming fault-tolerant, distributed, real-time applications. What differentiates it from most other languages is that it's a concurrent programming language; concurrency belongs to the language, not to the operating system. Its programs are collections of parallel processes cooperating to solve a particular problem that can be created quickly and have only limited memory

overhead; programmers can create large numbers of Erlang processes yet ignore any preconceived ideas they might have about limiting the number of processes in their solutions.

All Erlang processes are isolated from one another and in principle are “thread safe.” When Erlang applications are deployed on multicore computers, the individual Erlang processes are spread over the cores, and programmers do not have to worry about the details. The isolated processes share no data, and polymorphic messages can be sent between processes. In supporting strong isolation between processes and polymorphism, Erlang could be viewed as extremely object-oriented though without the usual mechanisms associated with traditional OO languages.

Erlang has no mutexes, and processes cannot share memory.^a Even within a process, data is immutable. The sequential Erlang subset that executes within an individual process is a dynamically typed functional programming language with immutable state.^b Moreover, instead of classes, methods, and inheritance, Erlang has modules that contain functions, as well as higher-order functions. It also includes processes, sophisticated error handling, code-replacement mechanisms, and a large set of libraries.

Here, I outline the key design criteria behind the language, showing how they are reflected in the language itself, as well as in programming language technology used since 1985.

Shared Nothing

The Erlang story began in mid-1985 when I was a new employee at the Ericsson Computer Science Lab in Stock-

^a The shared memory is hidden from the programmer. Practically all application programmers never use primitives that manipulate shared memory; the primitives are intended for writing special system processes and not normally exposed to the programmer.

^b This is not strictly true; processes can mutate local data, though such mutation is discouraged and rarely necessary.

68 COMMUNICATIONS OF THE ACM | SEPTEMBER 2010 | VOL. 53 | NO. 9



Summary

- Concurrency and memory models
- Transactions
- Message Passing

Coroutines and Asynchronous Programming

Advanced Systems Programming (H/M)
Lecture 8

Lecture Outline

- Motivation
- Coroutines, `async`, and `await`
- Design patterns for asynchronous code

Motivation

- How to overlap I/O and computation?
 - Multi-threading
 - Non-blocking I/O and `select()`
- Is there a better way?

Blocking I/O

```
fn read_exact<T: Read>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..])?;
    }
}
```

- **Desirable to perform I/O concurrently to other operations**
 - I/O operations are slow
 - Need to wait for network, disk, etc. – operations can take millions of cycles
 - I/O operations block the thread
 - Disrupts the user experience and prevents other computations
 - Want to overlap I/O and computation
 - Want to allow multiple concurrent I/O operations

Blocking I/O using Multiple Threads (1/2)

- Traditionally solved by moving blocking operations into separate threads:
 - Spawn dedicated threads to perform I/O operations concurrently
 - Re-join main thread/pass back result as message once complete
- Advantages:
 - Simple
 - No new language or runtime features
 - Don't have to change the way we do I/O
 - Do have to move I/O to a separate thread, communicate and synchronise
 - Concurrent code can run in parallel if the system has multiple cores
 - Safe, if using Rust, due to ownership rules preventing data races

```
fn main() {  
    ...  
    let (tx, rx) = channel();  
    thread::spawn(move|| {  
        ...perform I/O...  
        tx.send(results);  
    });  
    ...  
    let data = rx.recv();  
    ...  
}
```

Blocking I/O using Multiple Threads (2/2)

- Traditionally solved by moving blocking operations into separate threads:
 - Spawn dedicated threads to perform I/O operations concurrently
 - Re-join main thread/pass back result as message once complete
- Disadvantages:
 - Complex
 - Requires partitioning the application into multiple threads
 - Resource heavy
 - Each thread has its own stack
 - Context switch overheads
 - Parallelism offers limited benefits for I/O
 - Threads performing I/O often spend majority of time blocked
 - Wasteful to start a new thread that spends most of its time doing nothing

```
fn main() {  
    ...  
    let (tx, rx) = channel();  
    thread::spawn(move|| {  
        ...perform I/O...  
        tx.send(results);  
    });  
    ...  
    let data = rx.recv();  
    ...  
}
```

Non-blocking I/O and Polling (1/4)

- Blocking I/O using threads is problematic:
 - Threads provide concurrent I/O abstraction, but with high overhead
 - Multithreading **can** be inexpensive → Erlang
 - But has high overhead on general purpose operating systems
 - Higher context switch overhead due to security requirements
 - Higher memory overhead due to separate stack
 - Higher overhead due to greater isolation, preemptive scheduling
 - Limited opportunities for parallelism with I/O bound code
 - Threads **can** be scheduled in parallel, but to little benefit unless CPU bound

Non-blocking I/O and Polling (2/4)

- Lightweight alternative: multiplex I/O operations within a single thread
 - I/O operations complete asynchronously – why have threads block for them?
 - Provide a mechanism to start asynchronous I/O and poll the kernel for I/O events – all within a single application thread
 - Start an I/O operation
 - Periodically poll for progress of the I/O operation
 - If new data is available, a send operation has completed, or an error has occurred, then invoke the handler for that operation

Non-blocking I/O and Polling (3/4)

- Mechanisms for polling I/O for readiness
 - Berkeley Sockets API `select()` function in C
 - Or higher-performance, but less portable, variants such as `epoll` (Linux/Android), `kqueue` (FreeBSD/macOS/iOS), I/O completion ports (Windows)
 - Libraries such as `libevent`, `libev`, or `libuv` – common API for such system services
 - Rust `mio` library
- Key functionality:
 - Trigger non-blocking I/O operations: `read()` or `write()` to files, sockets, etc.
 - Poll kernel to check for readable or writeable data, or if errors are outstanding
 - Efficient and only requires a single thread, but requires code restructuring to avoid blocking

Non-blocking I/O and Polling (4/4)

- Berkeley Sockets API **select()** function in C:

```
FD_ZERO(&rfds);
FD_SET(fd1, &rfds);
FD_SET(fd2, &rfds);

tv.tv_sec = 5; // Timeout
tv.tv_usec = 0;

int rc = select(1, &rfds, &wfds, &efds, &tv);
if (rc < 0) {
    ... handle error
} else if (rc == 0) {
    ... handle timeout
} else {
    if (FD_ISSET(fd1, &rfds)) {
        ... data available to read() on fd1
    }
    if (FD_ISSET(fd2, &rfds)) {
        ... data available to read() on fd2
    }
    ...
}
```

select() polls a set of file descriptors for their readiness to **read()**, **write()**, or to deliver errors

FD_ISSET() checks particular file descriptor for readiness after **select()**

Low-level API well-suited to C programming; other libraries/languages provide comparable features

Alternatives to Non-blocking I/O?

- Non-blocking I/O can be highly efficient
 - Single thread can handle multiple I/O sources (sockets, file descriptors) at once
 - But – requires significant re-write of application code
 - Non-blocking I/O
 - Polling of I/O sources
 - Re-assembly of data
- Can we get the efficiency of non-blocking I/O in a more usable manner?
 - Maybe – coroutines and asynchronous code

Motivation

- How to overlap I/O and computation?
 - Multi-threading
 - Non-blocking I/O and `select()`
- Is there a better way?

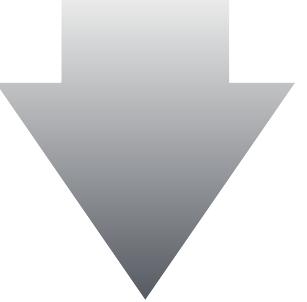
async and await

- Coroutines and asynchronous code
- Runtime support requirements

Coroutines and Asynchronous Code

- Aims to provide language and run-time support for I/O multiplexing on a single thread, in a more natural style

```
fn read_exact<T: Read>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..])?;
    }
}
```



```
async fn read_exact<T: AsyncRead>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..]).await?;
    }
}
```

- Runtime schedules **async** functions on a thread pool, yielding to other code on **await** calls → low-overhead concurrent I/O

Programming Model

- Structure I/O-based code as a set of concurrent *coroutines* that accept data from I/O sources and yield in place of blocking

What is a coroutine?

A generator **yields** a sequence of values:

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

>>> for i in countdown(5):
...     print i,
...
5 4 3 2 1
>>>
```

A function that can repeatedly run, yielding a sequence of values, while maintaining internal state

Calling **countdown(5)** produces a *generator object*. The **for** loop protocol calls **next()** on that object, causing it to execute until the next **yield** statement and return the yielded value.

→ Heap allocated; maintains state; executes only in response to external stimulus

Based on: <http://www.dabeaz.com/coroutines/Coroutines.pdf>

Programming Model

- Structure I/O-based code as a set of concurrent *coroutines* that accept data from I/O sources and yield in place of blocking

What is a coroutine?

A coroutine more generally consumes and yields values:

```
def grep(pattern):
    print(F"Looking for {pattern}")
    while True:
        line = (yield)
        if pattern in line:
            print line

>>> g = grep("python")
>>> g.next()
Looking for python
>>> g.send("Yeah, but no, but yeah, but no")
>>> g.send("A series of tubes")
>>> g.send("python generators rock!")
python generators rock!
>>>
```

The coroutines executes in response to **next()** or **send()** calls

Calls to **next()** make it execute until it next call **yield** to return a value

Calls to **send()** pass a value into the coroutine, to be returned by **(yield)**

Based on: <http://www.dabeaz.com/coroutines/Coroutines.pdf>

Programming Model

- Structure I/O-based code as a set of concurrent *coroutines* that accept data from I/O sources and yield in place of blocking

What is a coroutine?

A coroutine is a function that executes *concurrently* to – but not in parallel with – the rest of the code

It is event driven, and can accept and return values

Programming Model

- Structure I/O-based code as a set of concurrent *coroutines* that accept data from I/O sources and yield in place of blocking
 - An **async** function is a coroutine
 - Blocking I/O operations are labelled in the code – **await** – and cause control to pass to another coroutine while the I/O is performed
 - Provides concurrency without parallelism
 - Coroutines operate concurrently, but typically within a single thread
 - **await** passes control to another coroutine, and schedules a later wake-up for when the awaited operation completes
 - Encodes down to a state machine with calls to **select()**, or similar
 - Mimics structure of code with multi-threaded I/O – within a single thread

async Functions

- An **async** function is one that can act as a coroutine
 - It is executed *asynchronously* by the runtime
 - Widely supported – Python 3, JavaScript, C#, Rust, ...

```
#!/usr/bin/env python3

import asyncio

async def fetch_html(url: str, session: ClientSession) -> str:
    resp = await session.request(method="GET", url=url)
    html = await resp.text()
    return html
...
```

async tag on function

yield → **await**

But essentially a coroutine

- Main program must trigger asynchronous execution by the runtime:

```
asyncio.run(async function)
```

- Starts asynchronous polling runtime, runs until specified **async** function completes
- Runtime drives **async** functions to completion and handles switching between coroutines

await Future Results

- An **await** operation yields from the coroutine
 - Triggers I/O operation – and adds corresponding file descriptor to set polled by the runtime
 - Puts the coroutine in queue to be woken by the runtime, when file descriptor becomes ready

```
#!/usr/bin/env python3

import asyncio

async def fetch_html(url: str, session: ClientSession) -> str:
    resp = await session.request(method="GET", url=url)
    html = await resp.text()
    return html

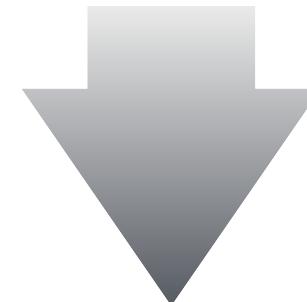
...
```

- If another coroutine is ready to execute then schedule wake-up once the I/O completes, and pass control passes to the other coroutine; else runtime blocks until either this, or some other, I/O operation becomes ready
- At some later time the file descriptor becomes ready and the runtime reschedules the coroutine – the I/O completes and the execution continues

async and await programming model

- Resulting asynchronous code should follow structure of synchronous (blocking) code:

```
fn read_exact<T: Read>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..])?;
    }
}
```



```
async fn read_exact<T: AsyncRead>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..]).await?;
    }
}
```

- Annotations (**async**, **await**) indicate asynchrony, context switch points
 - Compiler and runtime work together to generate code that can be executed in fragments when I/O operations occur

Runtime Support

- Asynchronous code needs runtime support to execute the coroutines and poll the I/O sources for activity
- An **async** function that returns data of type **T** compiles to a regular function that returns **impl Future<Output=T>**

```
pub trait Future {  
    type Output;  
    fn poll(self: Pin<&mut Self>, lw: &LocalWaker) -> Poll<Self::Output>;  
}
```

```
pub enum Poll<T> {  
    Ready(T),  
    Pending,  
}
```

- i.e., it returns a **Future** value that represents a value that will become available later
- The runtime continually calls **poll()** on **Future** values until all are **Ready**
 - A future returns **Ready** when complete, **Pending** when blocked on **awaiting** I/O
 - Calling **tokio::run(future)** starts the runtime
- Well supported in Python and JavaScript – runtime for Rust is experimental: <https://tokio.rs/>

async and await

- Coroutines and asynchronous code
- Runtime support requirements

Design Patterns for Asynchronous Code

- Compose **Future** values
- Avoid blocking I/O
- Avoid long-running computations

Compose Future Values

- **async** functions should be small, limited scope
- Perform a single well-defined task:
 - Read and parse a file
 - Read, process, and respond to a network request
- Rust provides combinators that can combine **Future** values, to produce a new **Future**:
 - **for_each()**, **and_then()**, **read_exact()**, **select()**
 - Can ease composition of asynchronous functions – but can also obfuscate

Avoid Blocking Operations

- Asynchronous code multiplexes I/O operations on single thread
 - Provides asynchronous aware versions of I/O operations
 - File I/O, network I/O (TCP, UDP, Unix sockets)
 - Non-blocking, return **Future** values that interact with the runtime
 - Does **not** interact well with blocking I/O
 - A **Future** that blocks on I/O will block **entire** runtime
- Programmer discipline required to ensure asynchronous and blocking I/O are not mixed within a code base
 - Including within library functions, etc.

Read → **AsyncRead**
Write → **AsyncWrite**

Avoid Long-running Computations

- Control passing between **Future** values is explicit
 - **await** calls switch control back to the runtime
 - Next runnable **Future** is then scheduled
 - A **Future** that doesn't call **await**, and instead performs some long-running computation, will starve other tasks
- Programmer discipline required to spawn separate threads for long-running computations
 - Communicate with these via message passing – that can be scheduled within a **Future**

An Asynchronous Future?

When to use Asynchronous I/O?

- **async/await** restructure code to efficiently multiplex large numbers of I/O operations on a single thread
 - Gives a **very natural programming model when each task is I/O bound**
 - Code to perform asynchronous, non-blocking, I/O is structured very similarly to code that uses blocking I/O operations
 - Runtime can schedule many tasks can run concurrently on a single thread
 - Each task is largely blocked awaiting I/O – little overheads

When to use Asynchronous I/O?

- **async/await** restructure code to efficiently multiplex large numbers of I/O operations on a single thread
 - **Problematic** with blocking operations
 - If a task performs a blocking operation, it locks the entire runtime – all potentially blocking calls must be updated to use asynchronous I/O operations
 - Potential to fragment the library ecosystem
 - **Problematic** with long-running computations
 - Long-running computations starve other tasks of CPU time – runtime only switches between tasks when an asynchronous operation is started
 - Need to insert context switch calls – isn't this just **cooperative multitasking** reimaged?
 - Windows 3.1, MacOS System 7

Performance of Asynchronous I/O

- Do you **really** need asynchronous I/O?
 - Threads are more expensive than **async** functions and tasks in a runtime
 - But threads are not **that** expensive – a properly configured modern machine can run thousands of threads
 - ~2,200 threads running on the Core i5 laptop these slides were prepared on, in normal use
 - Varnish web cache (<https://varnish-cache.org>): “it’s common to operate with 500 to 1000 threads minimum” but they “rarely recommend running with more than 5000 threads”
 - Unless you’re doing something **very** unusual you can likely just spawn a thread, or use a pre-configured thread pool, and perform blocking I/O – and communicate using channels, **even if this means spawning thousands of threads**
- Asynchronous I/O **can** give a performance benefit
 - But this performance benefit will usually be small
 - Choose asynchronous programming because you prefer the programming style, accepting that it will often not significantly improve performance

Summary

- Blocking I/O
 - Multi-threading → overheads
 - `select()` → complex
 - Coroutines and asynchronous code
- Is it worth it?

Security Considerations

Advanced Systems Programming (H/M)
Lecture 9



Lecture Outline

- Memory Safety
- Parsing and LangSec
- Modern Type Systems and Security

Memory Safety

- What is memory safety?
- Undefined behaviour in C and C++
- Security impact of memory un-safety
- Mitigations

Memory Safety in Programming Languages

- A **memory safe** language is one that ensures that the only memory accessed is that owned by the program, and that such access is done in a manner consistent of the declared type of the data
 - The program can access memory through its global and local variables, or through explicit references to them
 - The program can access heap memory it allocated via an explicit reference to that memory
 - All accesses obey the type rules of the language
- **Memory unsafe** languages fail to prevent accesses that break the type rules
 - C and C++ declare such behaviour to be **undefined**, but do nothing to prevent it from occurring

Memory Safe	Memory Unsafe
Java, Scala, C#, Rust, Go, Python, Ruby, Tcl, FORTRAN, COBOL, Modula-2, Occam, Erlang, Ada, Pascal, Haskell, ...	C, C++, Objective-C

Undefined Behaviour (1/13)

- What types of memory unsafe behaviour can occur in C and C++?

Undefined Behaviour (2/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation

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

A call to `malloc()` does not check if the amount of memory allocated corresponds to the size required to store the type

A memory safe language will require the size of the allocation to match the size of the allocated type at compile time

Operation ought to be “allocate enough memory for an object of type `T` and return a reference-to-`T`” and not “allocate n bytes of memory and return an untyped reference”

Undefined Behaviour (3/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - **Use before allocation**

```
char *buffer;

int rc = recv(fd, buffer, BUFLEN, 0);
if (rc < 0) {
    perror("Cannot receive");
} else {
    // Handle data
}
```

Passes a pointer to the **buffer** to the **recv()** function, but forgets to **malloc()** the memory – or assumes it'll be allocated in **recv()**

Memory safe languages require that all references be initialised and refer to valid data – good C compilers warn about this too

Undefined Behaviour (4/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - Use before allocation
 - Use after explicit **free()**

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

int main() {
    char *x = malloc(14);
    sprintf(x, "Hello, world!");
    free(x);
    printf("%s\n", x);
}
```

Accesses memory that has been explicitly freed, and hence is no longer accessible

Automatic memory management eliminates this class of bug

Undefined Behaviour (5/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - Use before allocation
 - Use after explicit `free()`
 - Use after implicit free

```
int *foo() {  
    int n = 42;  
    return &n;  
}
```

Return reference to stack allocated memory that is used after the stack frame has been destroyed

Automatic memory management eliminates this class of bug

Undefined Behaviour (6/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - Use before allocation
 - Use after explicit `free()`
 - Use after implicit free
 - Use of memory as the wrong type

```
char *buffer = malloc(BUFLEN);

int rc = recv(fd, buffer, BUFLEN, 0);
if (rc < 0) {
    ...
} else {
    struct pkt *p = (struct pkt *) buffer;
    if (p->version != 1) {
        ...
    }
    ...
}
```

Common in C code to see casts from `char *` buffers to more specific types (e.g., to a `struct` representing a network packet format)

Efficient – no memory copies – but unsafe since makes assumptions of `struct` layout in memory, size of block being cast

Memory safe language disallow arbitrary casts – write conversion functions instead; eliminates undefined behaviour

Undefined Behaviour (7/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - Use before allocation
 - Use after explicit `free()`
 - Use after implicit free
 - Use of memory as the wrong type
 - Use of string functions on non-string values

```
// Send the requested file
while ((rlen = read(inf, buf, BUFSIZE)) > 0) {
    if (send_response(fd, buf, strlen(buf)) == -1) {
        return -1;
    }
}
```

Strings in C are zero terminated, but `read()` does not add a terminator; buffer overflow results

Memory safe languages apply string bounds checks – runtime exception; safe failure, no undefined behaviour

Undefined Behaviour (8/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - Use before allocation
 - Use after explicit `free()`
 - Use after implicit free
 - Use of memory as the wrong type
 - Use of string functions on non-string values
 - **Heap allocation overflow**

```
#define BUFSIZE 256
int main(int argc, char *argv[]) {
    char *buf;
    buf = malloc(sizeof(char) * BUFSIZE);
    strcpy(buf, argv[1]);
}
```

Memory safe languages apply bounds checks to heap allocated memory – runtime exception; safe failure, no undefined behaviour

Undefined Behaviour (9/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - Use before allocation
 - Use after explicit `free()`
 - Use after implicit free
 - Use of memory as the wrong type
 - Use of string functions on non-string values
 - Heap allocation overflow
 - **Array bounds overflow**

```
int main(int argc, char *argv[]) {
    char buf[256];
    strcpy(buf, argv[1]);
}
```

Memory safe languages apply array bounds checks – runtime exception; safe failure, no undefined behaviour

Undefined Behaviour (10/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - Use before allocation
 - Use after explicit `free()`
 - Use after implicit free
 - Use of memory as the wrong type
 - Use of string functions on non-string values
 - Heap allocation overflow
 - Array bounds overflow
 - Arbitrary pointer arithmetic

`sizeof()` returns size in bytes, arithmetic on `int *` pointers is on pointer sized values; bounds check is too lax

Memory safe languages disallow arbitrary pointer arithmetic

```
int buf[INTBUFSIZE];
int *buf_ptr = buf;

while (havedata() && (buf_ptr < (buf + sizeof(buf)))) {
    *buf_ptr++ = parseint(getdata());
}
```

Undefined Behaviour (11/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - Use before allocation
 - Use after explicit **free()**
 - Use after implicit free
 - Use of memory as the wrong type
 - Use of string functions on non-string values
 - Heap allocation overflow
 - Array bounds overflow
 - Arbitrary pointer arithmetic
 - **Use of uninitialized memory**

```
static char *
read_headers(int fd)
{
    char      buf[BUFSIZE];
    char     *headers = malloc(1);
    size_t   headerLen = 0;
    ssize_t  rlen;

    while (strstr(headers, "\r\n\r\n") == NULL) {
        rlen = recv(fd, buf, BUFSIZE, 0);
        if (rlen == 0) {
            // Connection closed by client
            free(headers);
            return NULL;
        } else if (rlen < 0) {
            free(headers);
            perror("Cannot read HTTP request");
            return NULL;
        } else {
            headerLen += (size_t) rlen;
            headers = realloc(headers, headerLen + 1);
            strncat(headers, buf, (size_t) rlen);
        }
    }
    return headers;
}
```

Memory allocated with `malloc()` has undefined contents

Memory safe languages require memory to be initialised, or mandate that the runtime fills with known value

Undefined Behaviour (12/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - Type unsafe allocation
 - Use before allocation
 - Use after explicit `free()`
 - Use after implicit free
 - Use of memory as the wrong type
 - Use of string functions on non-string values
 - Heap allocation overflow
 - Array bounds overflow
 - Arbitrary pointer arithmetic
 - Use of uninitialised memory
 - Use of memory via dangling pointer
 - Use of memory via `null` pointer

`lookup()` call may fail, returning `null` pointer

Memory safe languages either: fail safely with an exception; or use `Option<>` types to enforce that the null pointer check is done

```
struct user *u = lookup(db, key);
printf("%s\n", u->name);
```

Undefined Behaviour (13/13)

- What types of memory unsafe behaviour can occur in C and C++?
 - ...and more <https://cwe.mitre.org/data/definitions/658.html>

Impact of Memory Unsafe Languages

- Lack of memory safety breaks the machine abstraction
 - With luck, program crashes – segmentation violation
 - If unlucky, memory unsafe behaviour corrupts other data owned by program
 - Undefined behaviour occurs
 - Cannot predict without knowing precise layout of program in memory
 - Difficult to debug
 - **Potential security risk** – corrupt program state to force arbitrary code execution

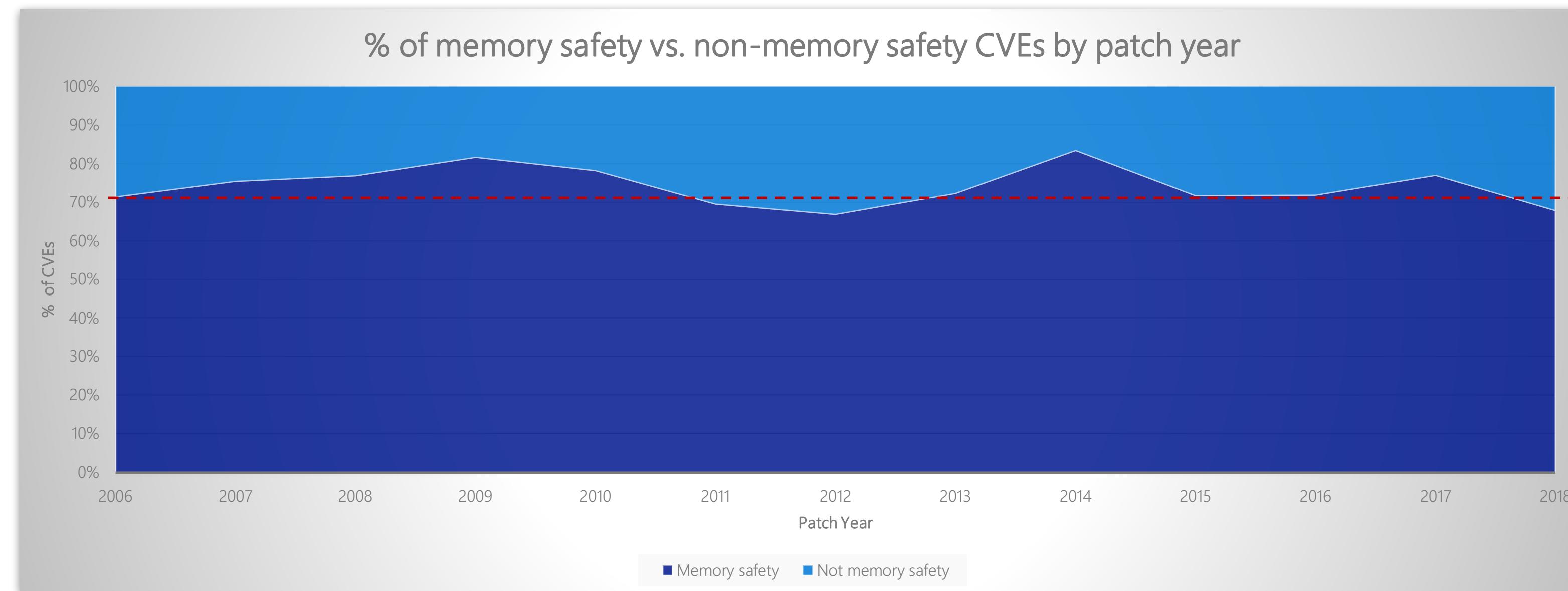
Impact of Memory Unsafe Languages – Security (1/2)

Vulnerabilities By Type																
Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits	
1999	894	177	112	172			2	7		25	16	103			2	
2000	1020	257	208	206			2	4	20		48	19	139			
2001	1677	403	403	297			7	34	123		83	36	220	2	2	
2002	2156	498	553	435	2	41	200	103		127	74	199	2	14	1	
2003	1527	381	477	371	2	49	129	60		1	62	69	144		16	5
2004	2451	580	614	410	3	148	291	111	12	145	96	134	5	38	5	
2005	4935	838	1627	657	21	604	786	202	15	289	261	221	11	100	14	
2006	6610	893	2719	663	91	967	1302	322		8	267	271	184	18	849	30
2007	6520	1101	2601	954	95	706	884	339	14	267	324	242	69	700	44	
2008	5632	894	2310	699	128	1101	807	363		7	288	270	188	83	170	74
2009	5736	1035	2185	700	188	963	851	322		9	337	302	223	115	138	738
2010	4652	1102	1714	680	342	520	605	275		8	234	282	238	86	73	1493
2011	4155	1221	1334	770	351	294	467	108		7	197	409	206	58	17	557
2012	5297	1425	1459	843	423	243	758	122		13	343	389	250	166	14	624
2013	5191	1455	1186	859	366	156	650	110		7	352	511	274	123	1	205
2014	7946	1598	1574	850	420	305	1105	204		12	457	2104	239	264	2	401
2015	6484	1791	1826	1079	749	218	778	150		12	577	748	367	248	5	127
2016	6447	2028	1494	1325	717	94	497	99		15	444	843	600	87	7	1
2017	14714	3154	3004	2805	745	503	1516	274		11	629	1706	459	327	18	6
2018	16555	1852	3035	2451	400	516	2001	509		11	709	1374	247	461	31	4
2019	4	2				1										
Total	110603	22685	30435	17226	5043	7438	13667	3823	162	5880	10104	4877	2123	2195	4333	
% Of All		20.5	27.5	15.6	4.6	6.7	12.4	3.5	0.1	5.3	9.1	4.4	1.9	2.0		

- Half of all security vulnerabilities are memory safety violations that should be caught by a modern type system
 - Buffer overflows
 - Memory corruption
 - Treating data as executable code
- Use of type-based modelling of the problem domain can help address others – by more rigorous checking of assumptions

Source: <https://www.cvedetails.com/vulnerabilities-by-types.php>

Impact of Memory Unsafe Languages – Security (2/2)



Source: Matt Miller, Microsoft, presentation at BlueHat IL conference, February 2019
https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL

- ~70% of Microsoft security updates fix bugs relating to unsafe memory usage
- This has not significantly changed in >10 years
- We're **not** getting better at writing secure code in memory unsafe languages

Mitigations for Memory Unsafe Languages (1/2)

- Use modern tooling for C and C++ development:
 - Compile C code with, at least, **clang -W -Wall -Werror**
 - Review documentation to find additional warnings it makes sense to enable
 - Fix **all** warnings – let the compiler help you debug your code
 - Use **clang** static analysis tools during debugging:
 - <https://clang.llvm.org/docs/AddressSanitizer.html>
 - <https://clang.llvm.org/docs/MemorySanitizer.html>
 - <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>
 - <https://clang.llvm.org/docs/ThreadSafetyAnalysis.html>
 - <https://clang.llvm.org/docs/ThreadSanitizer.html>
 - These have very high overhead, but catch many memory and thread safety problems

Mitigations for Memory Unsafe Languages (2/2)

- Use modern C++ language features:
 - I advise against C++ programming – language is too complex to understand
 - Too many features have been added over time, too few removed; code mixes old and new features
 - Very few people know the whole language
 - In hindsight, some C++ defaults were inappropriate
 - Modern C++ addresses many of these issues – but must retain backwards compatibility
 - Rust adopts many lessons learned in the development of C++
 - And can be simpler, since doesn't need worry about compatibility with legacy code
 - But if you have to use C++, modernise the code base, to use newer – safer – language features and idioms, where possible

Gradually Rewrite code into a Memory Safe Language

- Consider rewriting the most critical sections of the code in a memory safe language:
 - Gradually rewrite C into Rust
 - Rust can call C functions directly
 - Rust can be compiled into a library that can be directly linked with C or C++
 - <https://doc.rust-lang.org/nomicon/ffi.html>
 - <https://unhandledexpression.com/rust/2017/07/10/why-you-should-actually-rewrite-it-in-rust.html>
 - Possible to do a gradual rewrite of C or C++ into a safe language, while keeping the application usable and building at each stage
 - Gradually rewrite Objective-C into Swift
 - In the Apple ecosystem, similarly possible to gradually rewrite applications into Swift
 - **Difficult** and requires languages with compatible runtime models
 - Gradual rewrite likely has more chance of success than from-scratch rewrite

Memory Safety

- What is memory safety?
- Undefined behaviour in C and C++
- Security impact of memory un-safety
- Mitigations

Parsing and Network Security

- Postel's law
- Parsing

Remotely Exploitable Vulnerabilities

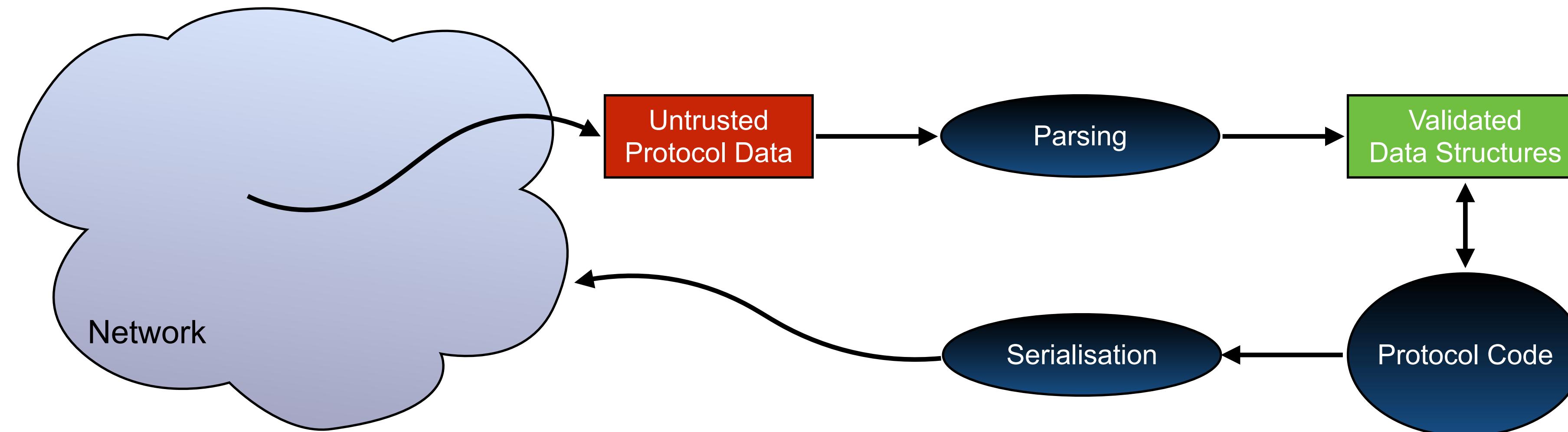
- Writing safe code in unsafe languages is difficult
 - Easy to rationalise each individual bug – “How could anyone write that?”

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
goto fail;
... other checks ...
fail:
    ... buffer frees (cleanups) ...
    return err;
```

<https://dwheelerauthor.com/essays/apple-goto-fail.html>

- But, people will continue to make mistakes
- Remotely exploitable vulnerabilities threaten any networked system
- **Are there particular classes of bug that cause such vulnerabilities?**

Parsing Untrusted Input



- The input parser is critical to security of a networked system
 - Takes untrusted input from the network
 - Generates validated, strongly typed, data structures that are processed by the rest of the code
- How can we ensure parsers are safe?

The Robustness Principle (Postel's Law)

At every layer of the protocols, there is a general rule whose application can lead to enormous benefits in robustness and interoperability:

"Be liberal in what you accept, and conservative in what you send"

Software should be written to deal with every conceivable error, no matter how unlikely; sooner or later a packet will come in with that particular combination of errors and attributes, and unless the software is prepared, chaos can ensue. In general, it is best to assume that the network is filled with malevolent entities that will send in packets designed to have the worst possible effect. This assumption will lead to suitable protective design, although the most serious problems in the Internet have been caused by un-envisaged mechanisms triggered by low-probability events; mere human malice would never have taken so devious a course!

Traditional guidance when writing networked systems is expressed in Postel's law

RFC1122

The Robustness Principle (Postel's Law)

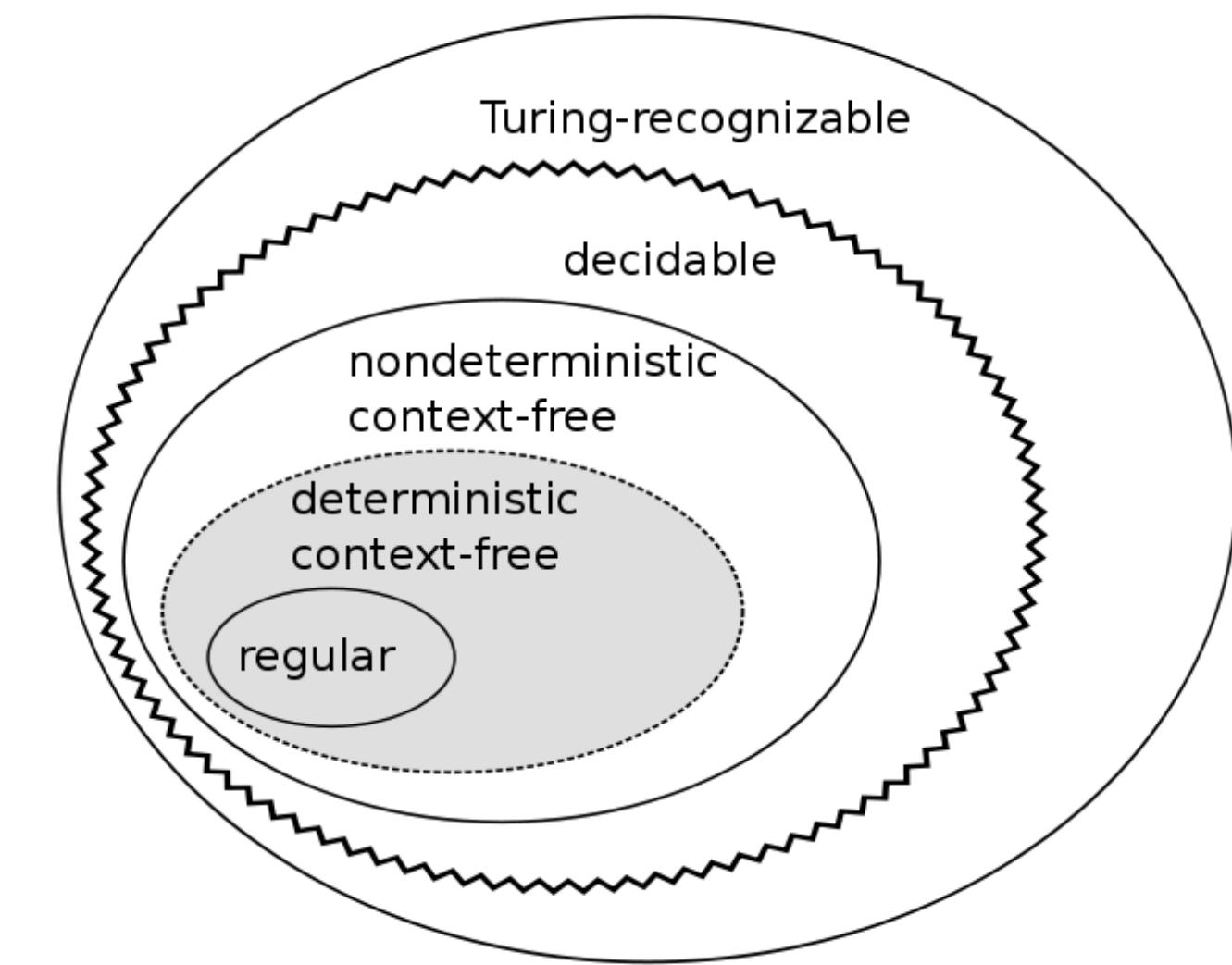
**"Be liberal in what you accept, and
conservative in what you send"**

“Postel lived on a network with all his friends.
We live on a network with all our enemies.
Postel was wrong for todays internet.”
— *Poul-Henning Kamp*

See also: <https://datatracker.ietf.org/doc/draft-iab-protocol-maintenance/>

Define Legal Inputs and Failure Responses

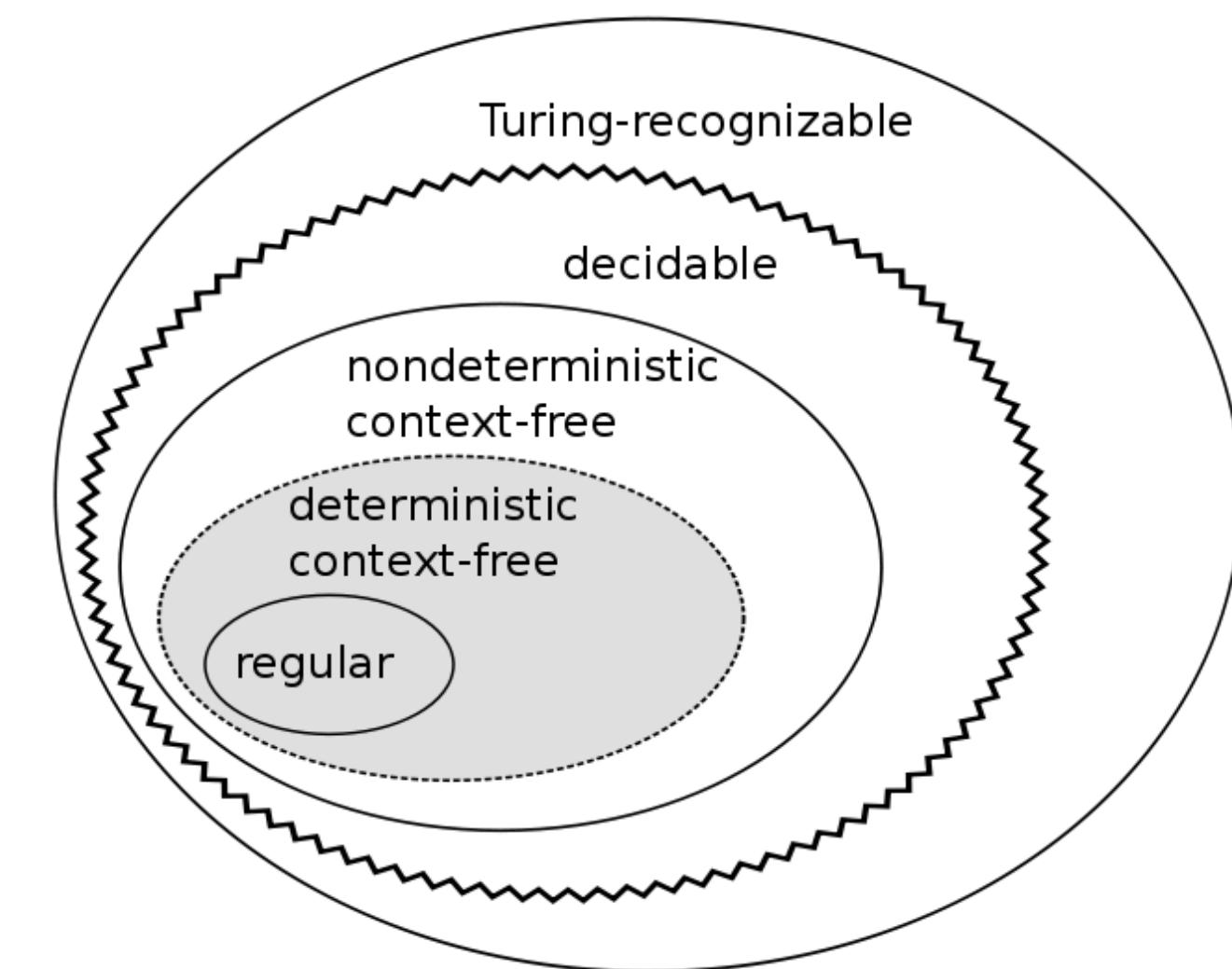
- The input parser takes untrusted input from the network, generates validated, strongly typed, data structures that are processed by the rest of the code
- A parser is an rule-driven automaton
 - It parses the input according to some grammar
 - If the input does not match the grammar, it fails
- **Formally specify the protocol grammar**
 - Define what is legal, and what is not, and write this down in a machine checkable manner
 - Define the grammar in as restrictive a manner as possible – e.g., don't use a Turing-complete parse when a regular expression will suffice
 - Specify what happens if the input data does not match the grammar: what causes a failure? How is it handled?



Source: <http://langsec.org/papers/Sassaman.pdf>

Generate the Parser (1/3)

- Difficult to reason about ad-hoc, manually written, parsing code
 - It performs low-level bit manipulation, string parsing, etc., all of which are hard to get correct
 - It tends to be poorly structured, hard to follow
- Rather, **auto-generate the parsing code:**
 - If input language is regular, use regular expression
 - If input language is context free, use a context free grammar
 - If neither of these work, use a more sophisticated parser, with minimal computational power
 - Generate strongly typed data structures, with explicit types to identify different data items
- Focus on parser correctness and readability
 - **Parsing performance matters less than security**



Generate the Parser (2/3)

- Use existing, well-tested, parser libraries
 - For Rust code, use **nom** or **combine**:

nom, eating data byte by byte

<https://github.com/Geal/nom>

Combine: A parser combinator library for Rust

<https://github.com/Marwes/combine>

- For C or C++, use **hammer**

Hammer: Parser combinator for binary formats, in C. Yes, in C. What? Don't look at me like that.

<https://github.com/UpstandingHackers/hammer>

The image shows a document page from a conference proceeding. At the top right, it says "2017 IEEE Symposium on Security and Privacy Workshops". The title of the paper is "Writing parsers like it is 2017". Below the title, two authors are listed: Pierre Chifflier from Agence Nationale de la Sécurité des Systèmes d'Information and Geoffroy Coutrie from Clever Cloud. The abstract discusses memory corruption bugs in C programs and how parser combinators can help prevent them. The paper is divided into several sections: I. INTRODUCTION, II. CURRENT SOLUTIONS, AND HOW TO GO FURTHER, A. Partial and bad solutions, and B. Reduce Damage. The text in the sections is technical, discussing software manipulation, security tools, and compiler functions.

P. Chifflier and G. Coutrie. Writing parsers like it is 2017.
IEEE Workshop on Language-Theoretic Security, San Jose,
CA, USA, May 2017. <https://dx.doi.org/10.1109/SPW.2017.39>

Generate the Parser (3/3)

```
# [derive(Debug, PartialEq, Eq)]
pub struct Header {
    pub version: u8,
    pub audio: bool,
    pub video: bool,
    pub offset: u32,
}
```

```
do_parse! (
    tag!("FLV") >>
    version: be_u8      >>
    flags:   be_u8      >>
    offset:  be_u32     >>
    (Header {
        version: version,
        audio:   flags & 4 == 4,
        video:   flags & 1 == 1,
        offset:  offset
    })
)
);
```

- Specify the types into which parsed data is stored
 - Describe the parser using an appropriate formal language – example uses `nom`
 - Generate the parser from that language
-
- Parsing is performed first, and either succeeds in its entirety or fails – the rest of the code uses only safe, pre-parsed, data

Source: P. Chifflier and G. Couprie. Writing parsers like it is 2017.
IEEE Workshop on Language-Theoretic Security, San Jose, CA,
USA, May 2017. <https://dx.doi.org/10.1109/SPW.2017.39>

Define Parsable Protocols

- If designing network protocols, consider ease of parsing
 - Minimise the amount of state and look-ahead required to parse the data
 - Prefer a predictable, regular, grammar to one that saves bits at the expense of complex parsing
 - Networks get faster, security vulnerabilities remain
 - The benefit of saving a few bits gets less over time
 - The benefit of being easy to parse securely remains

Parsing and LangSec

- **Read:** “The Bugs We Have to Kill”
 - Is this approach realistic?
 - Can we combine better parsing with modern, strongly typed, languages to improve network security?
 - Is performance good enough?
 - Can we improve the way we design protocols?

The Bugs We Have to Kill

SERGEY BRATUS, MEREDITH L. PATTERSON, AND ANNA SHUBINA

The code that parses inputs is the first and often the only protection for the rest of a program from malicious inputs. No programmer can afford to verify every implied condition on every line of code—even if this were possible to implement without slowing execution to a crawl. The parser is the part that is supposed to create a world for the rest of the program where all these implied conditions are true and need not be explicitly checked at every turn. Sadly, this is exactly where most parsers fail, and the rest of the program fails with them. In this article, we explain why parsers continue to be such a problem, as well as point to potential solutions that can kill large classes of bugs.

To do so, we are going to look at the problem from the computer science theory angle. Parsers, being input-consuming machines, are quite close to the theory’s classic computing models, each one an input-consuming machine: finite automata, pushdown automata, and Turing machines. The latter is our principal model of general-purpose programming, the computing model with the ultimate power and flexibility. Yet this high-end power and flexibility come with a high price, which Alan Turing demonstrated (and to whose proof we owe our very model of general-purpose programming): our inability to predict, by any general static analysis algorithm, how programs for it will execute.

Yet most of our parsers are just a layer on top of this totally flexible computing model. It is not surprising, then, that without carefully limiting our parsers’ design and code to much simpler models, we are left unable to prove these input-consuming machines secure. This is a powerful argument for making parsers and their input formats and protocols simpler, so that securing them does not require having to solve undecidable problems!

PARSERS, PARSERS EVERYWHERE

To quote Koprowski and Binsztok [1]:

Parsing is of major interest in computer science. Classically discovered by students as the first step in compilation, parsing is present in almost every program which performs data-manipulation. For instance, the Web is built on parsers. The HyperText Transfer Protocol (HTTP) is a parsed dialog between the client, or browser, and the server. This protocol transfers pages in HyperText Markup Language (HTML), which is also parsed by the browser. When running web-applications, browsers interpret JavaScript programs which, again, begins with parsing. Data exchange between browser(s) and server(s) uses languages or formats like XML and JSON. Even inside the server, several components (for instance the trio made of the HTTP server Apache, the PHP interpreter and the MySQL database) often manipulate programs and data dynamically; all require parsers.

So do the lower layers of the network stack down to the IP and the link layer protocols, and also other OS parts such as the USB drivers [2] (and even the hardware: turning PHY layer symbol streams into frames is parsing, too). For all of these core protocols, we add, their parsers have had a long history of failures, resulting in an Internet where any site, program, or system that receives untrusted input can be presumed compromised.

4 :login: AUGUST 2015 VOL. 40, NO. 4

www.usenix.org

S. Bratus, M. L. Patterson, and A. Shubina. The bugs we have to kill. ;login:, 40(4):4–10, August 2015. <http://langsec.org/papers/the-bugs-we-have-to-kill.pdf>

Parsing and Network Security

- Postel's law
- Parsing

Modern Type Systems and Security

- Make assumptions explicit
- Eliminate undefined behaviour

Causes of Security Vulnerabilities

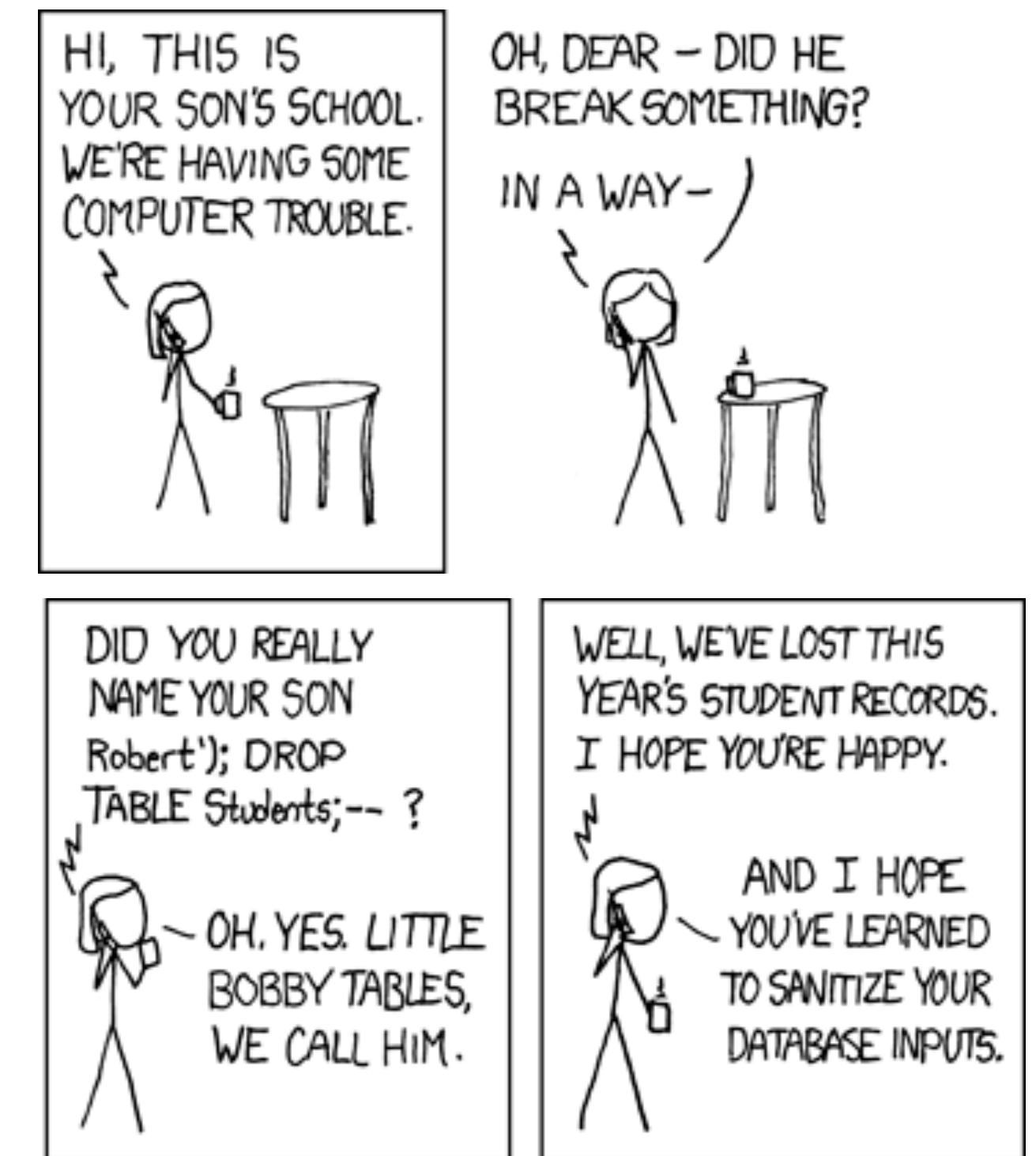
- Security vulnerabilities generally caused by persuading a program to do something that the programmer did not expect
 - Write past the end of a buffer
 - Treat user input as executable
 - Confuse a permission check
 - ...
- Violate an assumption in the code

Causes of Security Vulnerabilities

- Strong typing makes assumptions explicit
 - Use explicit types rather than generic types
 - Define safe conversion functions
 - Use phantom types where necessary, to apply semantic tags to data

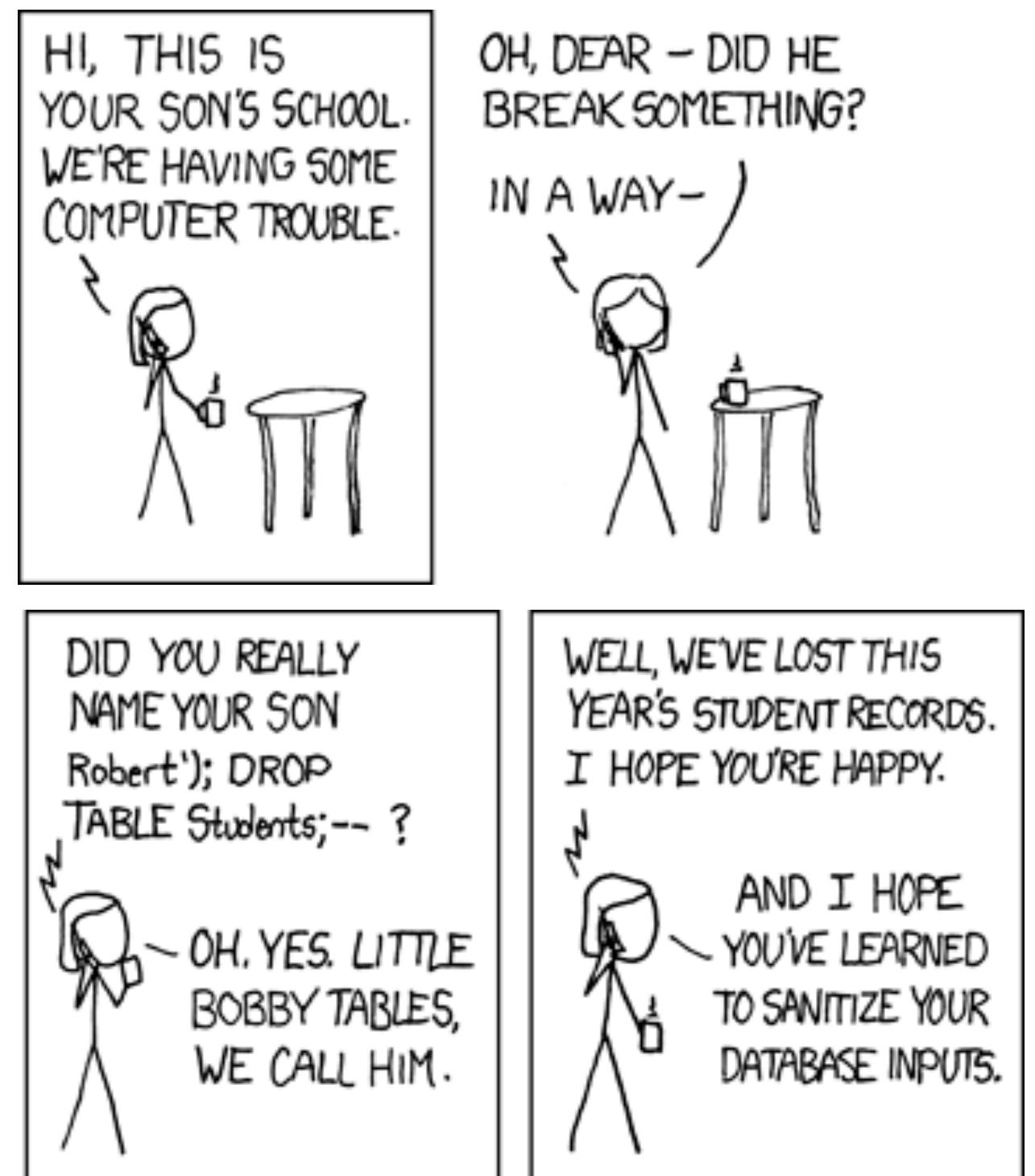
Prefer Explicit Types

- Vulnerabilities come from inconsistent data processing:
 - e.g., passing un-sanitised user entered data to a function that expects valid SQL
 - A **StudentName** and an **SqlString** are different; give them different types so compiler can catch inconsistent usage
 - Certain characters must be escaped before an arbitrary student name can be safely stored in an SQL database
- If **String** used everywhere, the programmer must manually check for consistency
 - Easy to make mistakes
 - If all the types are the same, the compiler can't help



Convert Data Carefully

- Explicit types require type conversions
 - Enforce security boundaries
 - Untrusted user input → escaped, safe, internal form; validate input before using it
 - Ensure only legal conversions occur



Use Phantom Types to Add Semantic Tags

- A phantom type parameter is one that doesn't show up at runtime, but is checked at compile time
 - In Rust, a **struct** with no fields has a type but is zero sized
 - Useful as type parameters to add semantic tags to data:

```
struct UserInput; // As received from the user
struct Sanitised; // After HTML codes have been escaped

fn sanitise_html(html : Html<UserInput>) -> Html<Sanitised> {
    ...
}
```

- Useful to represent states in a state machine

No Silver Bullet

- Memory safety and strong typing won't eliminate security vulnerabilities
- But, used carefully, they eliminate certain classes of vulnerability, and make others less likely by making hidden assumptions visible



Liability and Ethics

ACM code of ethics and professional conduct:

1.2 Avoid harm.

In this document, "harm" means negative consequences, especially when those consequences are significant and unjust. Examples of harm include unjustified physical or mental injury, unjustified destruction or disclosure of information, and unjustified damage to property, reputation, and the environment. This list is not exhaustive.

Well-intended actions, including those that accomplish assigned duties, may lead to harm. When that harm is unintended, those responsible are obliged to undo or mitigate the harm as much as possible. Avoiding harm begins with careful consideration of potential impacts on all those affected by decisions. When harm is an intentional part of the system, those responsible are obligated to ensure that the harm is ethically justified. In either case, ensure that all harm is minimized.

To minimize the possibility of indirectly or unintentionally harming others, computing professionals should follow generally accepted best practices unless there is a compelling ethical reason to do otherwise. Additionally, the consequences of data aggregation and

Security vulnerabilities and software failures routinely cause harm – can you justify your professional practice?

<https://www.acm.org/binaries/content/assets/about/acm-code-of-ethics-and-professional-conduct.pdf>

Summary

- Memory safety
- Parsing and LangSec
- Modern Type Systems and Security
- Ethics and Liability

Future Directions

Advanced Systems Programming (H/M)
Lecture 10



Systems Programming

- Course has discussed advanced topics in systems programming:
 - What is systems programming?
 - Type systems
 - Type-based modelling and design
 - Resource ownership and memory management
 - Garbage collection
 - Concurrency
 - Coroutines and asynchronous programming
 - Security considerations

Key Lessons

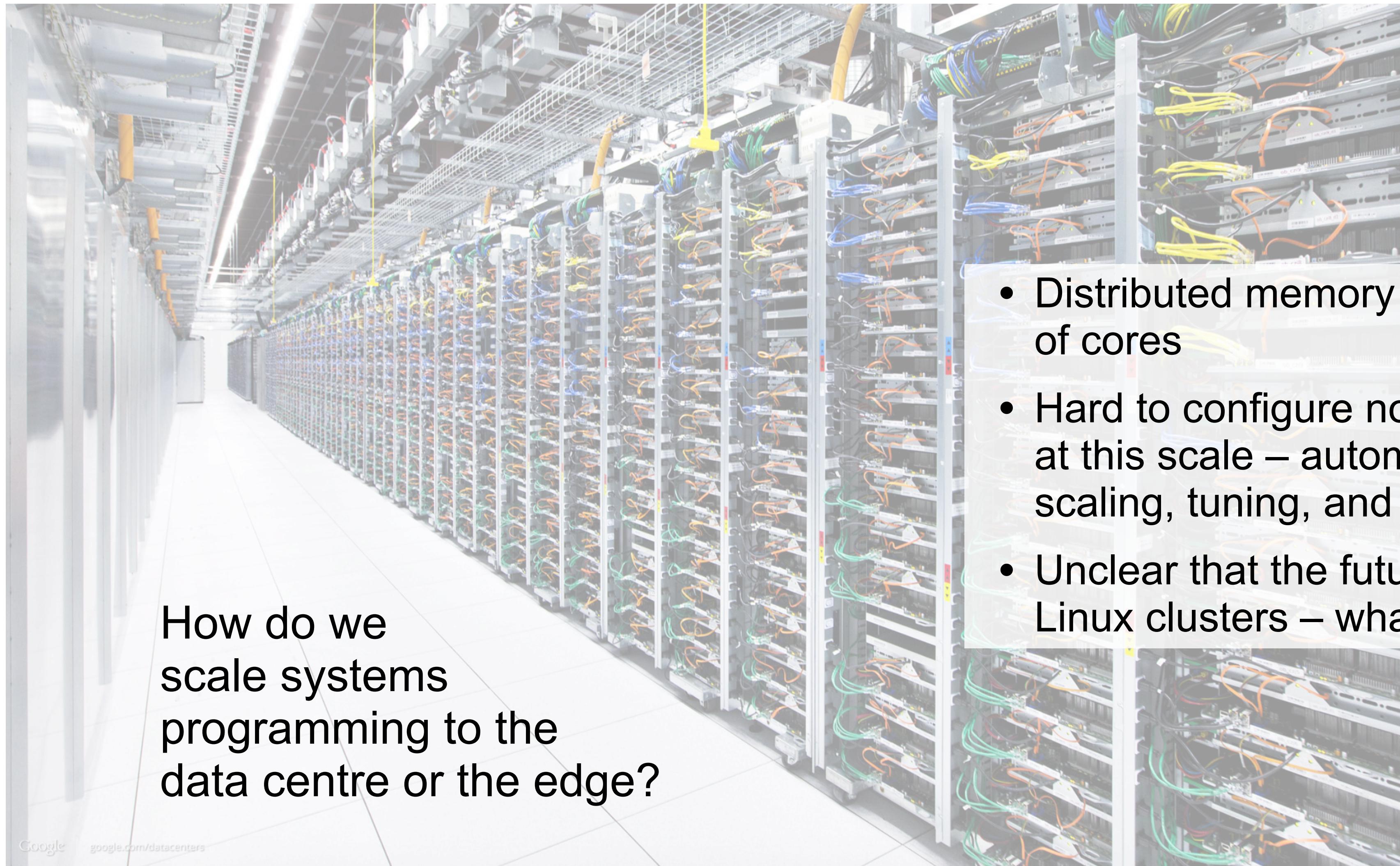
- Systems programs need low-level control of data layout
 - Device drivers
 - Network protocols
 - File I/O
- This control does not need to come at the expense of safety – modern languages can provide control
 - Memory safety with control over data layout
 - Avoidance of use-after-free, data races, iterator invalidation
- Safe concurrency is important, but all approaches have trade-offs
- Type systems can help model the problem domain
 - Model the assumptions – correctness and security

Future Directions

Concurrency

- Still no good solutions for concurrent programming
 - Transactional memory doesn't sit well with impure imperative languages
 - Message passing has issues with back pressure, race conditions, deadlocks, large-scale orchestration
 - Asynchronous code introduces subtle blocking and starvation bugs
 - Unclear what is the right solution
- None of these approaches scale to GPU programming → implicit parallelism? data parallel array types?

Data Centre Scale Computing



- Distributed memory computation on millions of cores
- Hard to configure nodes and communications at this scale – automatic configuration, scaling, tuning, and fault tolerance essential
- Unclear that the future should be large-scale Linux clusters – what comes next?

Distributed Operating Systems

- Barrelyfish – a message passing kernel for multicore systems
 - Each core runs a separate operating systems instance
 - All communication between cores is via message passing; no “main” CPU to act as central orchestrator
- A research prototype to test an idea, not a product
 - Where is the boundary for a Barrelyfish-like system?
 - Distinction between a distributed multi-kernel and a distributed system of networked computers? Should there be such a distinction?
 - How does it relate to concurrent programming, data centres, distributed edge computing, etc?

The Multikernel: A new OS architecture for scalable multicore systems

Andrew Baumann*, Paul Barham†, Pierre-Evariste Dagand‡, Tim Harris†, Rebecca Isaacs†,
Simon Peter*, Timothy Roscoe*, Adrian Schüpbach*, and Akhilesh Singhania*

*Systems Group, ETH Zurich

†Microsoft Research, Cambridge

‡ENS Cachan Bretagne

Abstract

Commodity computer systems contain more and more processor cores and exhibit increasingly diverse architectural tradeoffs, including memory hierarchies, interconnects, instruction sets and variants, and IO configurations. Previous high-performance computing systems have scaled in specific cases, but the dynamic nature of modern client and server workloads, coupled with the impossibility of statically optimizing an OS for all workloads and hardware variants pose serious challenges for operating system structures.

We argue that the challenge of future multicore hardware is best met by embracing the networked nature of the machine, rethinking OS architecture using ideas from distributed systems. We investigate a new OS structure, the *multikernel*, that treats the machine as a network of independent cores, assumes no inter-core sharing at the lowest level, and moves traditional OS functionality to a distributed system of processes that communicate via message-passing.

We have implemented a multikernel OS to show that the approach is promising, and we describe how traditional scalability problems for operating systems (such as memory management) can be effectively recast using messages and can exploit insights from distributed systems and networking. An evaluation of our prototype on multicore systems shows that, even on present-day machines, the performance of a multikernel is comparable with a conventional OS, and can scale better to support future hardware.

1 Introduction

Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for OS designers.

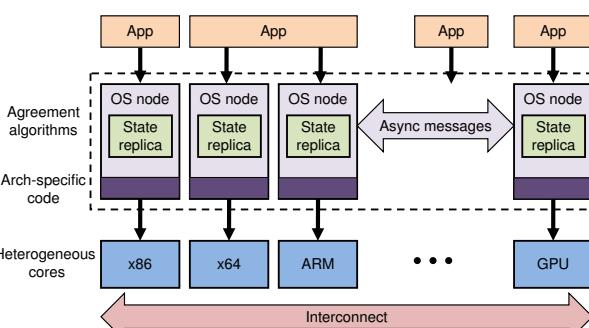


Figure 1: The multikernel model.

Such hardware, while in some regards similar to earlier parallel systems, is new in the general-purpose computing domain. We increasingly find multicore systems in a variety of environments ranging from personal computing platforms to data centers, with workloads that are less predictable, and often more OS-intensive, than traditional high-performance computing applications. It is no longer acceptable (or useful) to tune a general-purpose OS design for a particular hardware model: the deployed hardware varies wildly, and optimizations become obsolete after a few years when new hardware arrives.

Moreover, these optimizations involve tradeoffs specific to hardware parameters such as the cache hierarchy, the memory consistency model, and relative costs of local and remote cache access, and so are not portable between different hardware types. Often, they are not even applicable to future generations of the same architecture. Typically, because of these difficulties, a scalability problem must affect a substantial group of users before it will receive developer attention.

We attribute these engineering difficulties to the basic structure of a shared-memory kernel with data structures protected by locks, and in this paper we argue for rethinking the structure of the OS as a distributed system of functional units communicating via explicit mes-

1

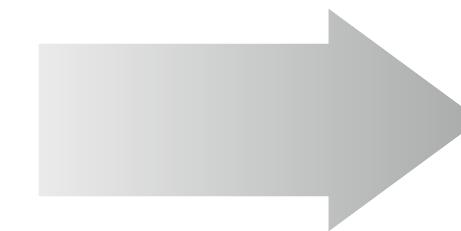
A. Baumann *et al.*, “The Multikernel: A new OS architecture for scalable multicore systems”, ACM Symposium on Operating Systems Principles, 2009. DOI:10.1145/1629575.1629579

Future Systems Programming Languages

- Increasingly seeing research ideas incorporated into mainstream programming languages
 - Rust, Swift – abstraction, resource management, strong and expressive type systems, low-level control
 - Erlang, Go – concurrency, fault tolerance, communication
- Choose the language based on problem domain, available tooling, and expertise
 - Good solutions for programming single systems
 - We still have a lot to learn about scaling to large clusters



Scaling Systems Programming



- Unix and C made sense when programming simple single-core systems
 - And form the basis of most existing systems – **but what comes next?**
- Modern massively concurrent and distributed applications need new approaches:
 - Low-level performance still crucial, but also need type systems and abstractions to hide the complexity – *no-one* can hold all the details in their head
 - Tooling is essential – to check invariants, document assumptions, ensure consistency
 - Deployment, configuration, and management must become increasingly autonomic – DevOps, infrastructure as code, self-managing

Wrap-Up

Assessment

- Marks for assessed coursework will be available shortly
- Final exam, worth 80% of the marks for the course, will be held in April/May
 - Sample exam paper, with worked answers, is available
 - Material from the lectures, labs, and cited papers is examinable
 - Aim is to test your understanding of the material, not to test your memory of all the details
 - Explain why – don't just recite what
- No specific revision lecture – use the Teams chat, or email, for revision questions

The End

;login: logout

The Night Watch

JAMES MICKENS



James Mickens is a researcher in the Distributed Systems group at Microsoft's Redmond lab. His current research focuses on web applications, with an emphasis on the design of JavaScript frameworks that allow developers to diagnose and fix bugs in widely deployed web applications. James also works on fast, scalable storage systems for datacenters. James received his PhD in computer science from the University of Michigan, and a bachelor's degree in computer science from Georgia Tech. mickens@microsoft.com

As a highly trained academic researcher, I spend a lot of time trying to advance the frontiers of human knowledge. However, as someone who was born in the South, I secretly believe that true progress is a fantasy, and that I need to prepare for the end times, and for the chickens coming home to roost, and fast zombies, and slow zombies, and the polite zombies who say "sir" and "ma'am" but then try to eat your brain to acquire your skills. When the revolution comes, I need to be prepared; thus, in the quiet moments, when I'm not producing incredible scientific breakthroughs, I think about what I'll do when the weather forecast inevitably becomes RIVERS OF BLOOD ALL DAY EVERY DAY. The main thing that I ponder is who will be in my gang, because the likelihood of post-apocalyptic survival is directly related to the size and quality of your rag-tag group of associates. There are some obvious people who I'll need to recruit: a locksmith (to open doors); a demolitions expert (for when the locksmith has run out of ideas); and a person who can procure, train, and then throw snakes at my enemies (because, in a world without hope, snake throwing is a reasonable way to resolve disputes). All of these people will play a role in my ultimate success as a dystopian warlord philosopher. However, the most important person in my gang will be a systems programmer. A person who can debug a device driver or a distributed system is a person who can be trusted in a Hobbesian nightmare of breathtaking scope; a systems programmer has seen the terrors of the world and understood the intrinsic horror of existence. The systems programmer has written drivers for buggy devices whose firmware was implemented by a drunken child or a sober goldfish. The systems programmer has traced a network problem across eight machines, three time zones, and a brief diversion into Amish country, where the problem was transmitted in the front left hoof of a mule named Deliverance. The systems programmer has read the kernel source, to better understand the deep ways of the universe, and the systems programmer has seen the comment in the scheduler that says "DOES THIS WORK LOL," and the systems programmer has wept instead of LOLED, and the systems programmer has submitted a kernel patch to restore balance to The Force and fix the priority inversion that was causing MySQL to hang. A systems programmer will know what to do when society breaks down, because the systems programmer already lives in a world without law.

;login: logout | NOVEMBER 2013 | WWW.USENIX.ORG

PAGE 5

J. Mickens. The night watch. *;login: logout*, pages 5–8, November 2013.
https://www.usenix.org/system/files/1311_05-08_mickens.pdf