Next Steps in Systems Programming

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

BY ND
Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

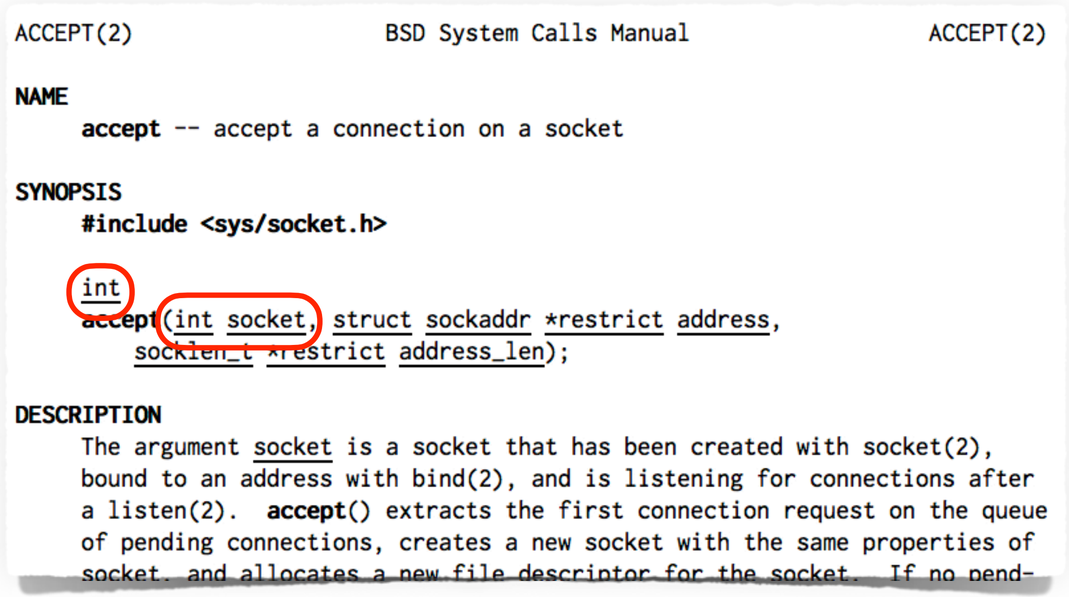
Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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 pend-

- 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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
  - Unsuited 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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);
}
```

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

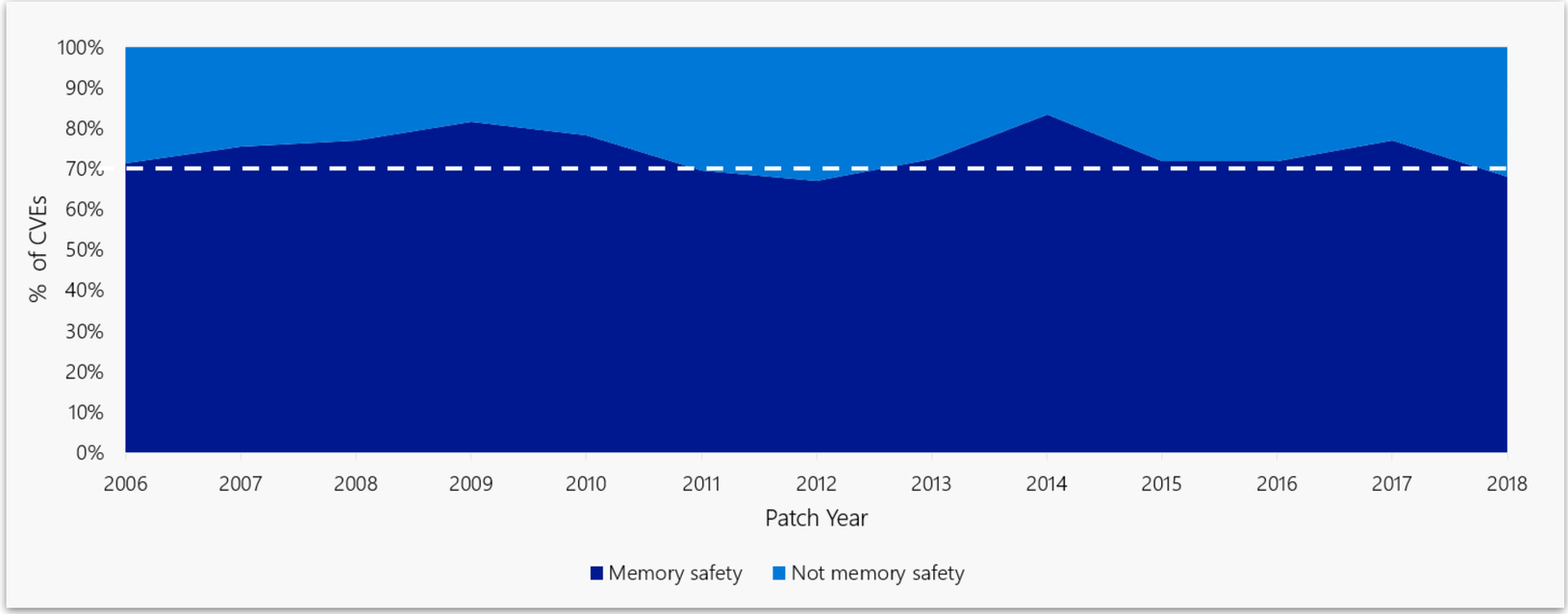
How to Improve Memory Management & Safety? (2/2)

- 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

How to Improve Security?



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

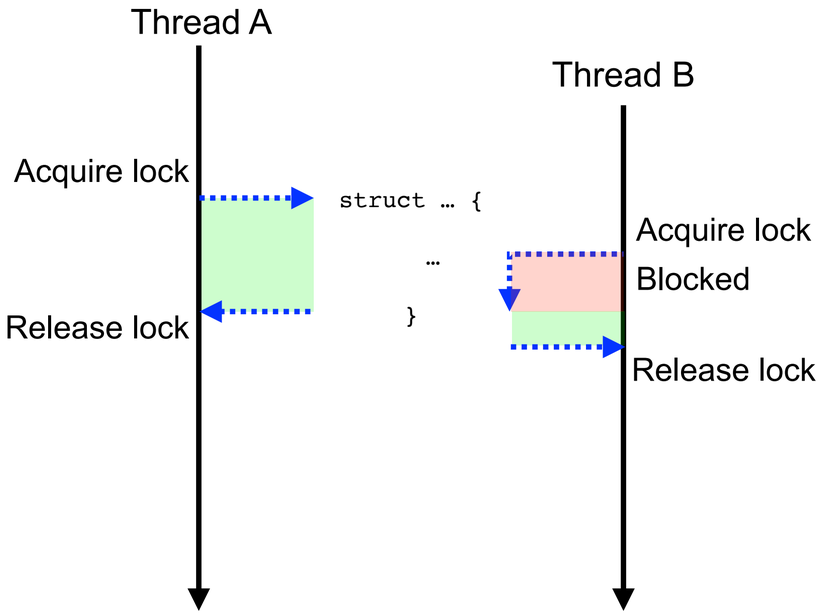
- ~70% of all reported security vulnerabilities are memory safety violations that should be caught by a modern type system – buffer overflows, use-after-free, memory corruption, treating data as executable code
- Use of type based modelling of the problem domain can help address others – by more rigorous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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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**

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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**

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Summary

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow