Do We Know Enough About Memory?

Nandakumar Edamana

March 3, 2022

Question

Which piece do you like? Which one the computer would?

Question

Where does the variable *n* reside? RAM, Hard Disk, or somewhere else?

```
int main()
{
        int n = 10;
        sleep(60 * 15);

        return n;
}
```

Question

Is this safe?

```c
int main()
{
        char passwd[16];

        printf("Enter the password: ");
        scanf("%s", passwd);

        return 0;
}
```

Why Bother

Reasons

Why bother when we have (i) modern hardware and (ii) high-level languages?

Why Bother

Reasons

Why bother when we have (i) modern hardware and (ii) high-level languages?

Performance Memory hierarchy, cache locality, etc.

Why Bother

Reasons

Why bother when we have (i) modern hardware and (ii) high-level languages?

Performance Memory hierarchy, cache locality, etc.

Economy Buying a PC, renting a server, etc.

Why Bother

Reasons

Why bother when we have (i) modern hardware and (ii) high-level languages?

Performance	Memory hierarchy, cache locality, etc.
Economy	Buying a PC, renting a server, etc.
Limits	Hardware limits, memory leaks, etc.

Why Bother

Reasons

Why bother when we have (i) modern hardware and (ii) high-level languages?

Performance	Memory hierarchy, cache locality, etc.
Economy	Buying a PC, renting a server, etc.
Limits	Hardware limits, memory leaks, etc.
Security	Memory-related vulnerabilities

Why Bother

Reasons

Why bother when we have (i) modern hardware and (ii) high-level languages?

Performance	Memory hierarchy, cache locality, etc.
Economy	Buying a PC, renting a server, etc.
Limits	Hardware limits, memory leaks, etc.
Security	Memory-related vulnerabilities
Integrity	Bit flip, bit rot, etc.

Why Bother

Reasons

Why bother when we have (i) modern hardware and (ii) high-level languages?

Performance	Memory hierarchy, cache locality, etc.
Economy	Buying a PC, renting a server, etc.
Limits	Hardware limits, memory leaks, etc.
Security	Memory-related vulnerabilities
Integrity	Bit flip, bit rot, etc.
Environment	E-waste, carbon footprint

(All are equally important)

Why Bother

Do I Have Control?

- How deep can software get?
  - compiler flags, attributes, intrinsics, OS API, etc.
- How deep can HLLs like Python get?
  - control flow, GC tricks, external libraries, etc.

General

Bit Flip

[1] https://www.johndcook.com/blog/2019/05/20/cosmic-rays-flipping-bits/

[2] https://www.businessinsider.in/transportation/tech-companies-have-been-silently-battling-a-bizarre-phenomenon-called-cosmic-rays-that-would-otherwise-wreak-havoc-on-our-electronics/articleshow/70046484.cms

General

Bit Flip

- 2003 Belgian election cosmic ray bit flip claims [1][2]

[1] https://www.johndcook.com/blog/2019/05/20/cosmic-rays-flipping-bits/

[2] https://www.businessinsider.in/transportation/tech-companies-have-been-silently-battling-a-bizarre-phenomenon-called-cosmic-rays-that-would-otherwise-wreak-havoc-on-our-electronics/articleshow/70046484.cms

General
Bit Flip

- 2003 Belgian election cosmic ray bit flip claims [1][2]
  - Mismatch exactly by 4096 (single bit flip)

[1] https://www.johndcook.com/blog/2019/05/20/cosmic-rays-flipping-bits/
[2] https://www.businessinsider.in/transportation/tech-companies-have-been-silently-battling-a-bizarre-phenomenon-called-cosmic-rays-that-would-otherwise-wreak-havoc-on-our-electronics/articleshow/70046484.cms

General

Bit Flip

- 2003 Belgian election cosmic ray bit flip claims [1,2]
  - Mismatch exactly by 4096 (single bit flip)
- Electrical interference, malfunction, cosmic rays

---

[1] https://www.johndcook.com/blog/2019/05/20/cosmic-rays-flipping-bits/

[2] https://www.businessinsider.in/transportation/tech-companies-have-been-silently-battling-a-bizarre-phenomenon-called-cosmic-rays-that-would-otherwise-wreak-havoc-on-our-electronics/articleshow/70046484.cms

General

Bit Flip

- 2003 Belgian election cosmic ray bit flip claims [1,2]
    - Mismatch exactly by 4096 (single bit flip)
- Electrical interference, malfunction, cosmic rays
- Modern hardware is more vulnerable because they are small

---

[1] https://www.johndcook.com/blog/2019/05/20/cosmic-rays-flipping-bits/

[2] https://www.businessinsider.in/transportation/tech-companies-have-been-silently-battling-a-bizarre-phenomenon-called-cosmic-rays-that-would-otherwise-wreak-havoc-on-our-electronics/articleshow/70046484.cms

- 2003 Belgian election cosmic ray bit flip claims [1,2]
  - Mismatch exactly by 4096 (single bit flip)
- Electrical interference, malfunction, cosmic rays
- Modern hardware is more vulnerable because they are small
- ECC RAM

[1] https://www.johndcook.com/blog/2019/05/20/cosmic-rays-flipping-bits/
[2] https://www.businessinsider.in/transportation/tech-companies-have-been-silently-battling-a-bizarre-phenomenon-called-cosmic-rays-that-would-otherwise-wreak-havoc-on-our-electronics/articleshow/70046484.cms

- 2003 Belgian election cosmic ray bit flip claims [1][2]
    - Mismatch exactly by 4096 (single bit flip)
- Electrical interference, malfunction, cosmic rays
- Modern hardware is more vulnerable because they are small
- ECC RAM
    - What about other points of failure like CPU?

---

[1] https://www.johndcook.com/blog/2019/05/20/cosmic-rays-flipping-bits/

[2] https://www.businessinsider.in/transportation/tech-companies-have-been-silently-battling-a-bizarre-phenomenon-called-cosmic-rays-that-would-otherwise-wreak-havoc-on-our-electronics/articleshow/70046484.cms

General

Bit Rot

- Physical damage
- Obsolescence

What is Computer Memory?

Concepts

- What is a computer?
- Stored program, stored data, and temporary data
- Temporary data includes intermediate results, pointers like SP and IP, process info, etc.
- CPU cannot store everything

What is Computer Memory?

Primary vs Secondary

- Primary vs Secondary
- Primary

What is Computer Memory?

Primary vs Secondary

- Primary vs Secondary
- Primary
  - CPU registers

What is Computer Memory?

Primary vs Secondary

- Primary vs Secondary
- Primary
  - CPU registers
  - CPU cache

What is Computer Memory?

Primary vs Secondary

- Primary vs Secondary
- Primary
  - CPU registers
  - CPU cache
  - ROM

What is Computer Memory?

Primary vs Secondary

- Primary vs Secondary
- Primary
  - CPU registers
  - CPU cache
  - ROM
  - RAM
- Random Access vs Sequential Access
- Content-addressable

General

Units

[3] International Electrotechnical Commission

General

Units

- KB vs KiB (IEC [3] units)

[3] International Electrotechnical Commission

General

Units

- KB vs KiB (IEC [3] units)
- How many bits is a byte?

[3] International Electrotechnical Commission

General
Units

- KB vs KiB (IEC [3] units)
- How many bits is a byte?
- Nibble

[3] International Electrotechnical Commission

General

Units

- KB vs KiB (IEC [3] units)
- How many bits is a byte?
- Nibble
- Word

[3] International Electrotechnical Commission

n-bit Computing

Can mean:

n-bit Computing

Can mean:

- Word size is n bits

n-bit Computing

Can mean:

- Word size is n bits
- Addressing limit is 2^n bits

General

Browser Are Memory Hogs

General

Browser Are Memory Hogs

- Rich Web sites and apps

General

Browser Are Memory Hogs

- Rich Web sites and apps
- JavaScript memory leaks

Browser Are Memory Hogs

- Rich Web sites and apps
- JavaScript memory leaks
- Prefetching

General

Browser Are Memory Hogs

- Rich Web sites and apps
- JavaScript memory leaks
- Prefetching
- Electron-based apps

General

Memory Cleaning Apps

General

Memory Cleaning Apps

Ugly Malware

General

Memory Cleaning Apps

Ugly	Malware
Bad	Clears useful cache

General

Memory Cleaning Apps

Ugly	Malware
Bad	Clears useful cache
Good	Don't exist

Memory Leak

Textbook Style

```
main() {
        char * arr = malloc(1024);

        // Use arr, but don't free()

        arr = NULL; /* or some new allocation */
}
```

General

Before We Fork...

Some useful info related to secondary memory before we start focusing on primary memory:

[4] Self-Monitoring, Analysis and Reporting Technology

General

Before We Fork...

Some useful info related to secondary memory before we start focusing on primary memory:

- How to monitor memory and disk usage?

[4] Self-Monitoring, Analysis and Reporting Technology

General

Before We Fork...

Some useful info related to secondary memory before we start focusing on primary memory:

- How to monitor memory and disk usage?
- SMART [4]

[4] Self-Monitoring, Analysis and Reporting Technology

General

Before We Fork...

Some useful info related to secondary memory before we start focusing on primary memory:

- How to monitor memory and disk usage?
- SMART [4]
- RAID

[4] Self-Monitoring, Analysis and Reporting Technology

General

Before We Fork...

Some useful info related to secondary memory before we start focusing on primary memory:

- How to monitor memory and disk usage?
- SMART [4]
- RAID
- SSD

[4] Self-Monitoring, Analysis and Reporting Technology

Some useful info related to secondary memory before we start focusing on primary memory:

- How to monitor memory and disk usage?
- SMART [4]
- RAID
- SSD
- Filesystems

[4] Self-Monitoring, Analysis and Reporting Technology

General

Before We Fork...

Some useful info related to secondary memory before we start focusing on primary memory:

- How to monitor memory and disk usage?
- SMART [4]
- RAID
- SSD
- Filesystems
- Tape storage

[4] Self-Monitoring, Analysis and Reporting Technology

General

Before We Fork...

Some useful info related to secondary memory before we start focusing on primary memory:

- How to monitor memory and disk usage?
- SMART [4]
- RAID
- SSD
- Filesystems
- Tape storage
- tmpfs

[4] Self-Monitoring, Analysis and Reporting Technology

General

Before We Fork...

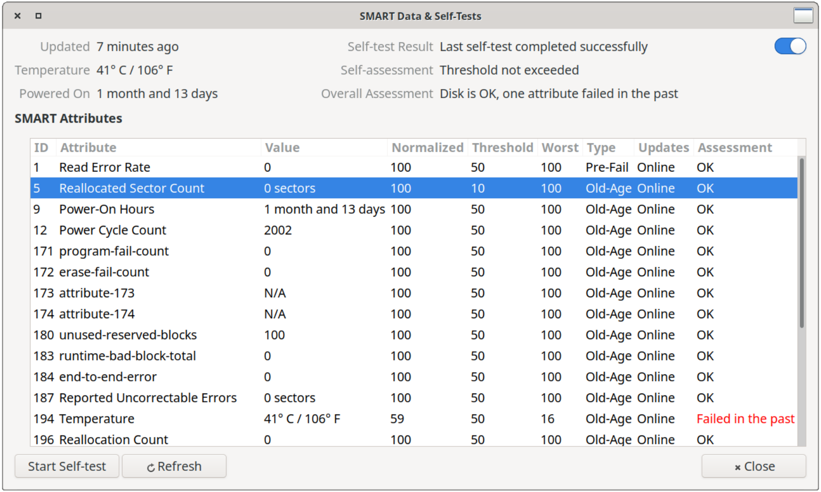
Some useful info related to secondary memory before we start focusing on primary memory:

- How to monitor memory and disk usage?
- SMART [4]
- RAID
- SSD
- Filesystems
- Tape storage
- tmpfs
- Hardware details: yes, we skip

[4] Self-Monitoring, Analysis and Reporting Technology

General

SMART



ID	Attribute	Value	Normalized	Threshold	Worst	Type	Updates	Assessment
1	Read Error Rate	0	100	50	100	Pre-Fail	Online	OK
5	Reallocated Sector Count	0 sectors	100	10	100	Old-Age	Online	OK
9	Power-On Hours	1 month and 13 days	100	50	100	Old-Age	Online	OK
12	Power Cycle Count	2002	100	50	100	Old-Age	Online	OK
171	program-fail-count	0	100	50	100	Old-Age	Online	OK
172	erase-fail-count	0	100	50	100	Old-Age	Online	OK
173	attribute-173	N/A	100	50	100	Old-Age	Online	OK
174	attribute-174	N/A	100	50	100	Old-Age	Online	OK
180	unused-reserved-blocks	100	100	50	100	Old-Age	Online	OK
183	runtime-bad-block-total	0	100	50	100	Old-Age	Online	OK
184	end-to-end-error	0	100	50	100	Old-Age	Online	OK
187	Reported Uncorrectable Errors	0 sectors	100	50	100	Old-Age	Online	OK
194	Temperature	41° C / 106° F	59	50	16	Old-Age	Online	Failed in the past
196	Reallocation Count	0	100	50	100	Old-Age	Online	OK

General

RAID

```
# mdadm -D /dev/md1
/dev/md1:
               ...
        Raid Level : raid1
        Array Size : 54271424 (51.76 GiB 55.57 GB)
               ...
       Update Time : Thu Mar  3 08:29:06 2022
             State : clean
               ...
              Name : nandakumar-laptop:md1  (local to host nandakumar-laptop)
               ...
    Number   Major   Minor   RaidDevice State
       2       8        2        0      active sync   /dev/sda2
       1       8       20        1      active sync   /dev/sdb4
```

General

Questions?

Memory Bugs

Memory Bugs

- Memory leak

Memory Bugs

- Memory leak
- Double free

Memory Bugs

- Memory leak
- Double free
- Dangling pointer

Memory Bugs

- Memory leak
- Double free
- Dangling pointer
- Null dereference

Memory Bugs

- Memory leak
- Double free
- Dangling pointer
- Null dereference
- Out-of-bound access

Memory Bugs

- Memory leak
- Double free
- Dangling pointer
- Null dereference
- Out-of-bound access
- Stack overflow (mainly due to recursion)

Memory Bugs

- Memory leak
- Double free
- Dangling pointer
- Null dereference
- Out-of-bound access
- Stack overflow (mainly due to recursion)
- CPU vulnerabilities like Spectre and Meltdown

Virtual Memory

Before That

Virtual Memory

Before That

- One program at a time

Virtual Memory

Before That

- One program at a time
- Unrestricted access to memory

Virtual Memory

Before That

- One program at a time
- Unrestricted access to memory
- Absolute addresses

Disclaimer: History is more complicated.

Virtual Memory

Characteristics

Virtual Memory

Characteristics

- The stage is mine

Virtual Memory
Characteristics

- The stage is mine
- Address space

Virtual Memory

Characteristics

- The stage is mine
- Address space
  - More than physical

Virtual Memory

Characteristics

- The stage is mine
- Address space
  - More than physical
  - Non-unique addresses

Virtual Memory

Characteristics

- The stage is mine
- Address space
  - More than physical
  - Non-unique addresses
- Memory protection

Virtual Memory

Characteristics

- The stage is mine
- Address space
  - More than physical
  - Non-unique addresses
- Memory protection
- Swap

Virtual Memory

Characteristics

- The stage is mine
- Address space
  - More than physical
  - Non-unique addresses
- Memory protection
- Swap

Disclaimer: Reality is more complicated.

- Pages
- Frames
- Swapping
- Page table and TLB
- Page fault
- Thrashing

Memory Management

Memory Management

- Manual

Memory Management

- Manual
- Automatic

Memory Management

- Manual
- Automatic
  - Garbage collection

Memory Management

- Manual
- Automatic
  - Garbage collection
  - Compile-time checks and allocation decisions

Memory Management

- Manual
- Automatic
  - Garbage collection
  - Compile-time checks and allocation decisions
  - Safety checks (null dereference, out-of bound access, etc.)

Memory Layout

Args and env

Stack

...

Heap

bss

data

text

Types of Allocation

Types of Allocation

Static Allocation Initialized global data (`.data`)

Types of Allocation

Static Allocation Initialized global data (`.data`)

Stack Allocation Compile-time allocation (mostly), runtime growth

Types of Allocation

Static Allocation Initialized global data (`.data`)

Stack Allocation Compile-time allocation (mostly), runtime growth

Heap Allocation Totally dynamic (runtime)

Types of Allocation

Static Allocation Initialized global data (`.data`)

Stack Allocation Compile-time allocation (mostly), runtime growth

Heap Allocation Totally dynamic (runtime)

NOTE: Stack allocation can also have dynamic nature (`alloca()`, for instance).

Types of Allocation

Stack and Heap

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

typedef struct {
        char * name;
        int    yob;
} Person;

void die_if_null(void * ptr) {
        if(!ptr) {
                perror(NULL);
                exit(1);
        }
}
```

Types of Allocation

Stack and Heap

```c
int main()
{
        int n; // stack allocation

        printf("Enter the no. of people (>0): ");
        scanf("%d", &n); // no new allocation

        Person * people = malloc(n * sizeof(Person)); // heap allocation
        die_if_null(people);

        for(int i = 0; i < n; i++) { // stack allocation
                const char * dob;

                printf("\nEnter the name of Person %d: ", i + 1);
                scanf("%ms", &(people[i].name)); // new heap allocation
                die_if_null(people[i].name);

                printf("Enter the YoB of Person %d:  ", i + 1);
                scanf("%d", &(people[i].yob)); // no new allocation
        }
```

Types of Allocation

Stack Growth

NOTE: Stack size is fixed and limited throughout the execution.

1. _start() → main()
2. main() → a()
   a() returns
3. main() → b()
4. b() → c()
   c() returns, b() returns, main() returns

```
_() -> m()      SP = 0x7ffedd207ad0
m() -> a()      SP = 0x7ffedd207ab0
a() returns     SP = 0x7ffedd207ad0
m() -> b()      SP = 0x7ffedd207ab0
b() -> c()      SP = 0x7ffedd207a90
c() returns     SP = 0x7ffedd207ab0
b() returns     SP = 0x7ffedd207ad0
```

Types of Allocation

How Stack Is Allocated

```
main:
    ...
            pushq    %rbp
    ...
            movq     %rsp, %rbp
    ...
            subq     $32, %rsp
```

Types of Allocation

How Stack Is Allocated

- Essentially a couple of simple instructions

Types of Allocation

How Stack Is Allocated

- Essentially a couple of simple instructions
- Allocates all variables in a function at once

Types of Allocation

How Stack Is Accessed

`i = 0`:

```
movl    $0, -20(%rbp)
```

Types of Allocation

How Stack Is Accessed

`i = 0`:

```
movl    $0, -20(%rbp)
```

- `i = 0` means *move 0 to i*

Types of Allocation
How Stack Is Accessed

`i = 0`:

```
movl    $0, -20(%rbp)
```

- `i = 0` means *move 0 to i*
- `i` is at *rbp - 20*

Types of Allocation

How Stack Is Accessed

`i = 0`:

```
movl    $0, -20(%rbp)
```

- `i = 0` means *move 0 to i*
- `i` is at *rbp - 20*
- So *move 0 to the location rbp - 20*

Types of Allocation

How Heap Is Allocated

```
call    malloc@PLT
movq    %rax, -16(%rbp)
```

Types of Allocation

How Heap Is Allocated

```
call    malloc@PLT
movq    %rax, -16(%rbp)
```

- malloc() calls other functions and performs expensive calculations

Types of Allocation

How Heap Is Allocated

```
call    malloc@PLT
movq    %rax, -16(%rbp)
```

- malloc() calls other functions and performs expensive calculations
- Each object or dynamic array needs its own malloc() call

Types of Allocation

How Heap Is Allocated

```
call    malloc@PLT
movq    %rax, -16(%rbp)
```

- malloc() calls other functions and performs expensive calculations
- Each object or dynamic array needs its own malloc() call
- Accessed using pointers

Types of Allocation

When to?

Types of Allocation

When to?

- Stack when the size is small and known in advance

Types of Allocation

When to?

- Stack when the size is small and known in advance
- Heap for large, size-unknown, or passed-around objects

Types of Allocation
When to?

- Stack when the size is small and known in advance
- Heap for large, size-unknown, or passed-around objects

NOTE: not all passed-around objects have to be heap-allocated.

Dynamic Memory Pitfalls

- free() is lazy
- malloc() may fail in future

Python Memory Management

Basics

Python Memory Management

Basics

- Private Heap

Python Memory Management

Basics

- Private Heap
- Reference-counted Garbage Collection

Python Memory Management

Basics

- Private Heap
- Reference-counted Garbage Collection
- Overallocation for dynamic data structures

Python Memory Management

Allocation

Objects are not cloned by default:

```
>>> class MyClass:
...     pass
...
>>> obj1 = MyClass()
>>> obj2 = MyClass()
>>> obj3 = obj1
>>> id(obj1)
140197270756992
>>> id(obj2)
140197270163072
>>> id(obj3)
140197270756992
>>>
```

Python Memory Management

Allocation

What is the difference between $=$ and $+=$?

```
>>> s = 'Hello'
>>> s = s + ', world'
>>> s += '!'
>>> s
'Hello, world!'
>>>
```

Python Memory Management

Allocation

$+=$ *might* perform in-place addition.

```
>>> s = 'Hello'
>>> id(s)
140336922823728
>>> s = s + ', world'
>>> s
'Hello, world'
>>> id(s)
140336922865776
>>> s += '!'
>>> s
'Hello, world!'
>>> id(s)
140336922865776
>>>
```

Python Memory Management

Allocation

$+=$ is in-place only if *__iadd__* method has been implemented.

So far, so good:

```
>>> s = 'Hello'
>>> id(s)
140253776977008
>>> s = s + ', world!'
>>> s
'Hello, world!'
>>> id(s)
140253777023088
>>>
```

So far, so good:

```
>>> s = 'Hello'
>>> id(s)
140253776977008
>>> s = s + ', world!'
>>> s
'Hello, world!'
>>> id(s)
140253777023088
>>>
```

Wait, what?

```
>>> s = 'Hello'
>>> id(s)
140253776977008
>>> s += ', world!'
>>> s
'Hello, world!'
>>> id(s)
140253777023088
>>>
```

Python Memory Management

Allocation

- The second part was run in the same session

Python Memory Management

Allocation

- The second part was run in the same session
- Strings are immutable in Python

Python Memory Management

Allocation

- The second part was run in the same session
- Strings are immutable in Python
- So the implementation *may* choose to pool them

Python Memory Management

Concatenation

From https://docs.python.org/3.8/library/stdtypes.html:

6. Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. To get a linear runtime cost, you must switch to one of the alternatives below:
   - if concatenating `str` objects, you can build a list and use `str.join()` at the end or else write to an `io.StringIO` instance and retrieve its value when complete
   - if concatenating `bytes` objects, you can similarly use `bytes.join()` or `io.BytesIO`, or you can do in-place concatenation with a `bytearray` object. `bytearray` objects are mutable and have an efficient overallocation mechanism
   - if concatenating `tuple` objects, extend a `list` instead
   - for other types, investigate the relevant class documentation

Python Memory Management

Strings Are Mutable?

How come $+=$ is possible if strings are immutable in Python?

Python Memory Management

Strings Are Mutable?

How come $+=$ is possible if strings are immutable in Python?

CPython $+=$ reuses the object only if the reference count is 1.

Go Memory Management

Stack vs Heap

Go Memory Management

Stack vs Heap

- There is stack and heap in practice

Go Memory Management

Stack vs Heap

- There is stack and heap in practice
- They are low-level unlike Python, but the selection happens automatically

Go Memory Management

Stack vs Heap

Important points from https://go.dev/doc/faq#stack_or_heap:

[5] This differs from C

Important points from https://go.dev/doc/faq#stack_or_heap:

- Whether on stack or heap doesn't affect correctness. "Each variable in Go exists as long as there are references to it."[5]

[5] This differs from C

Go Memory Management

Stack vs Heap

Important points from https://go.dev/doc/faq#stack_or_heap:

- Whether on stack or heap doesn't affect correctness. "Each variable in Go exists as long as there are references to it."[5]
- The storage location does have an effect on writing efficient programs.

[5] This differs from C

Go Memory Management

Stack vs Heap

Important points from https://go.dev/doc/faq#stack_or_heap:

- Whether on stack or heap doesn't affect correctness. "Each variable in Go exists as long as there are references to it."[5]
- The storage location does have an effect on writing efficient programs.
- Allocated on heap:

[5] This differs from C

Go Memory Management

Stack vs Heap

Important points from https://go.dev/doc/faq#stack_or_heap:

- Whether on stack or heap doesn't affect correctness. "Each variable in Go exists as long as there are references to it."[5]
- The storage location does have an effect on writing efficient programs.
- Allocated on heap:
  - if the compiler cannot prove that the variable is not referenced after the function returns OR

[5] This differs from C

Go Memory Management

Stack vs Heap

Important points from https://go.dev/doc/faq#stack_or_heap:

- Whether on stack or heap doesn't affect correctness. "Each variable in Go exists as long as there are references to it."[5]
- The storage location does have an effect on writing efficient programs.
- Allocated on heap:
  - if the compiler cannot prove that the variable is not referenced after the function returns OR
  - if a local variable is very large

[5] This differs from C

Go Memory Management

Stack vs Heap

Important points from https://go.dev/doc/faq#stack_or_heap:

- Whether on stack or heap doesn't affect correctness. "Each variable in Go exists as long as there are references to it."[5]
- The storage location does have an effect on writing efficient programs.
- Allocated on heap:
  - if the compiler cannot prove that the variable is not referenced after the function returns OR
  - if a local variable is very large
- "In the current compilers, if a variable has its address taken, that variable is a candidate for allocation on the heap. However, a basic escape analysis recognizes some cases when such variables will not live past the return from the function and can reside on the stack."

[5] This differs from C

Go Memory Management

new() vs make()

Go Memory Management

new() vs make()

new():

Go Memory Management

new() vs make()

new():

- Allocates

Go Memory Management

new() vs make()

new():

- Allocates
- Zeros-out

Go Memory Management

new() vs make()

new():

- Allocates
- Zeros-out
- Returns the pointer

Go Memory Management

new() vs make()

new():

- Allocates
- Zeros-out
- Returns the pointer

make() acts like a constructor for certain composite types, and returns the object itself.

Go Memory Management

Use of Virtual Memory

From https://go.dev/doc/faq:

> *Why does my Go process use so much virtual memory?*
> *The Go memory allocator reserves a large region of virtual memory as an arena for allocations. This virtual memory is local to the specific Go process; the reservation does not deprive other processes of memory.*
> *To find the amount of actual memory allocated to a Go process, use the Unix top command and consult the RES (Linux) or RSIZE (macOS) columns.*

https://go.dev/doc/faq#goroutines (talks about resizable stacks) https://go.dev/doc/codewalk/sharemem/

Important points from https://go.dev/ref/mem:

Go Memory Model

Serialization

Important points from https://go.dev/ref/mem:

- "The Go memory model specifies the conditions under which reads of a variable in one goroutine can be guaranteed to observe values produced by writes to the same variable in a different goroutine."

Go Memory Model

Important points from https://go.dev/ref/mem:

- "The Go memory model specifies the conditions under which reads of a variable in one goroutine can be guaranteed to observe values produced by writes to the same variable in a different goroutine."
- To serialize shared access, use channels or sync primitives (e.g.: the ones from the `sync` package)

Go Memory Model

Pitfall

"Incorrect synchronization" from https://go.dev/ref/mem:

> *Note that a read r may observe the value written by a write w that happens concurrently with r. Even if this occurs, it does not imply that reads happening after r will observe writes that happened before w.*

Go Memory Model

Pitfall

```
var a, b int

func f() {
        a = 1
        b = 2
}

func g() {
        print(b)
        print(a)
}

func main() {
        go f()
        g()
}
```

Go Memory Model

Pitfall

```go
var a, b int

func f() {
	a = 1
	b = 2
}

func g() {
	print(b)
	print(a)
}

func main() {
	go f()
	g()
}
```

We expect g to print 2 and 1 or 0 and 0.

Go Memory Model

Pitfall

```
var a, b int

func f() {
        a = 1
        b = 2
}

func g() {
        print(b)
        print(a)
}

func main() {
        go f()
        g()
}
```

We expect g to print 2 and 1 or 0 and 0.

But g may print 2 and then 0, not 2 and 1 or 0 and 0.

Memory Leak in GC Languages

Memory Leak in GC Languages

- Global/resident objects

Memory Leak in GC Languages

- Global/resident objects
- Memory leak in libraries

Memory Leak in GC Languages

- Global/resident objects
- Memory leak in libraries
- Not calling finalizers required by external libraries

Memory Leak in GC Languages

- Global/resident objects
- Memory leak in libraries
- Not calling finalizers required by external libraries
- Circular reference

Memory Leak in GC Languages

JavaScript

Memory Leak in GC Languages

JavaScript

- Important for SPAs and PWAs

Memory Leak in GC Languages

JavaScript

- Important for SPAs and PWAs
- From https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/:

Memory Leak in GC Languages

JavaScript

- Important for SPAs and PWAs
- From https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/:
  - Global variables

Memory Leak in GC Languages

JavaScript

- Important for SPAs and PWAs
- From https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/:
  - Global variables
  - Forgotten timers or callbacks

Memory Leak in GC Languages

JavaScript

- Important for SPAs and PWAs
- From https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/:
  - Global variables
  - Forgotten timers or callbacks
  - Out of DOM references

Memory Leak in GC Languages

JavaScript

- Important for SPAs and PWAs
- From https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/:
  - Global variables
  - Forgotten timers or callbacks
  - Out of DOM references
  - Closures

CPU Cache

Rationale

- Temporal Locality
- Spatial Locality

CPU Cache

Leverage

CPU Cache

Leverage

- Access arrays in sequential order

CPU Cache

Leverage

- Access arrays in sequential order
- Pack structures

CPU Cache

Leverage

- Access arrays in sequential order
- Pack structures
- Minimize data movement

CPU Cache

Leverage

- Access arrays in sequential order
- Pack structures
- Minimize data movement
- Avoid accessing the same cache block from different threads - (to avoid cache coherence overhead)

CPU Cache

Leverage

- Access arrays in sequential order
- Pack structures
- Minimize data movement
- Avoid accessing the same cache block from different threads - (to avoid cache coherence overhead)
- Avoid optimizations that cause cache misses (e.g.: loop unrolling, maybe)

Questions?

Thank You