Effizientes Programmieren in C, C++ und Rust

Karlsruher Institut für Technologie

Memory Allocations (in C), Preprocessor, and Branch Prediction

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024

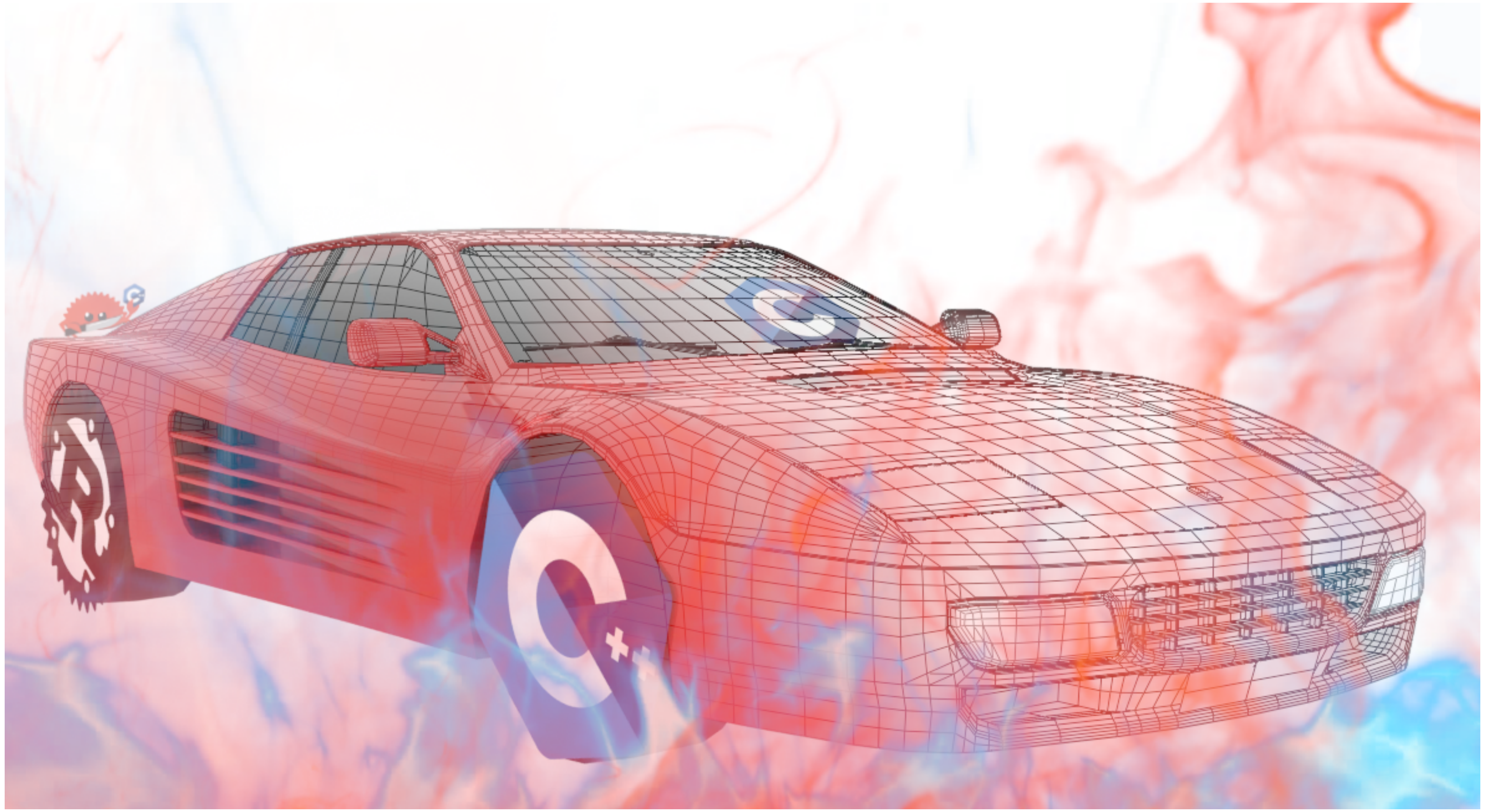


Table of Contents

Last Lecture

Pointing Out the Memory

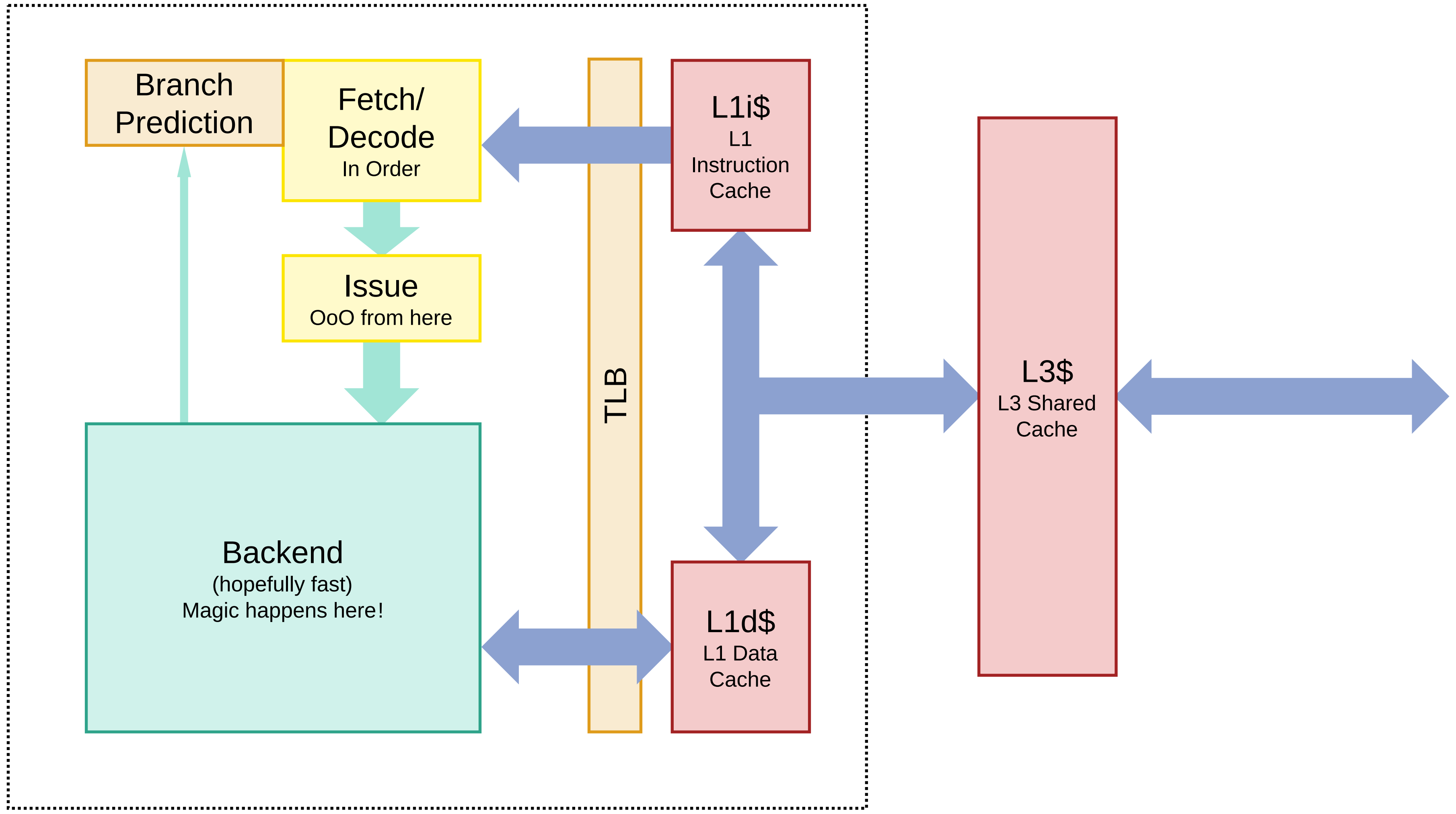
C RECAP

- Types
- Functions
- `struct`

MEMORY

- Pointers
- Caches
- Prefetchers

To Branch Or Not to Branch, That Is the Question



Again: µEffCpp

BRANCH PREDICTION

- Decides whether branch is predicted to be taken or not
- Learns from recent occurences
- Easiest case:
  - Branch upwards: taken
  - Branch downwards: not taken

  Why meaningful? Loops mostly repeat
- Second most easy example: Same behavior as last time
- Our model: Combines both
  - Unknown taken if upwards
  - Otherwise last behavior

Branch Prediction

Performance Impact 🚴

PREDICTION CORRECT

- CPU knew next instruction(s)
- CPU prefetched said instruction(s)
- CPU (partially) decoded & issued said instruction(s)
- CPU (partially) executed said instruction(s)
- CPU successfully verified the branch prediction
- Branch was basically free (except memory)

PREDICTION INCORRECT

- CPU expected other instructions
- CPU's queues (decode, issue, execute) are blocked by those instructions
- CPU wasted cycles on wrong work
- CPU cleans queues
- CPU fetches correct instructions
- Then: see "Prediction Correct"
- About 15-20 cycles of wrong work in queues + memory

Preprocessor

Glorified `sed 's/foo/bar/g'`

Preprocessor: Text replacement before actual compiler runs

Main uses

- Copy in text from other files
- Name text constants
- Name text replacement functions
- Conditionally insert/remove code

Preprocessor

This >> # << Is Not a Hashtag!

THE USES OF

- `#include`
- `#define`
- `#undef`
- `#if` / `#if defined` / `#ifdef`
- `#else` / `#elif`
- `#endif`
- `#error`
- `#warning`
- `#pragma`

Preprocessor

Lets Cook `#include <Copy_Pasta.h>`

`#include <FILENAME>`:

- removes the line
- instead inputs the verbatim contents of FILENAME

`#include <FILENAME>` vs
`#include "FILENAME"`?

- `<>`: *Usually* system / library files
- `""`: *Usually* files from current project

Whats in FILENAME?

- *Usually* used for headers

HEADERS?

Collects data to share code with

- Type definitions: What does this type look like?
- function / variable declarations: This element exists

Preprocessor

Access Violation: `#define private public`

`#define SOMETHING`: replace every (following) occurence of SOMETHING with something else

SIMPLE MACRO

`#define SOMETHING OTHERTHING`

- SOMETHING now called a macro
- SOMETHING can be (nearly) every character sequence
- Newlines need to be escaped using \ like this
- *Usually* used for constants
- Example: `#define PI 4`

MACRO "FUNCTION"

`#define SOMETHING(PARAM) OTHERTHING`

- SOMETHING can now use parameters
- parameters are verbatimly replaced
- Example: `#define MAX ((a)>(b)?(a):(b))`
- Example call: `MAX(i++, j++)`
- Replaced to: `((i++)>(j++)?(i++):(j++))`
  Attention! Modifies i and j

And `#undef SOMETHING`? makes preprocessor forget SOMETHING

Preprocessor

`#if I_COULD_FLY`

`#if SOMETHING`: Only insert if SOMETHING is != 0

- Preprocessor: No (explicit) parentheses
- End of `#if` directive: `#endif`
- Has `#else` and `#elif`
- Can be nested! (Comments cannot be nested)

```
#if 0
    #if 1
    // this comment dies
    #endif
#else
    #if 1
    // this comment survives
    #endif
#endif
```

`#if defined`

- Often: Do not care about macro value, just existance
- `defined(MACRONAME)`-Operator to the rescue
- Shorthand: `#ifdef`

Preprocessor

`#warning lecture new, expect mishaps`

`#error fatal`

- Handy if you know you cannot compile

```
#if (__STDC_VERSION__ < 201112L)
    #error this code requires C11 for multithreading
#endif
```

`#warning non fatal`

- Handy to communicate with programmer

```
#if defined(_MSC_VER)
    #warning microsoft compilers are untested
#endif
```

Not Preprocessor

```
#pragma GCC diagnostic ignored "-Wuninitialized"
```

`#pragma` is not (really) a preprocessor directive

Allocating Memory

Actually Initializing Our Vectors

STACK

- Local memory of function
- Cannot be `return`ed (!) (Cleaned up on scope exit)
- Passing into called function ok? Yes
- Fixed size per function
- 🚴Performance: Extremely hot, basically always in L1 Cache ⇒ use often

HEAP

- Global memory, shared
- Can be `return`
- Dynamically sized
- 🚴Performance: Depends on usage pattern

STATIC MEMORY

- "Forbidden" Category...
- Use with extreme caution
- 🚴Performance: Similar to heap

Stacks and Scopes

Where Variables Deserve to Die

```
int main() {
    struct ll_vector last = {0, 2};
    struct ll_vector mid = {&last, 1};
    { // explicit scope
        struct ll_vector first = {&mid, 0};
    } // first stops living here
} // last, mid stop living here
```

- Scope: "area" of common livetime of variables
- Lifetime: time of valid use
- Lifetime: Starts at definition, ends at enclosing scope close
- Scopes (in C): basically everything between { and }
- Scopes: functions, loop/if bodies
- Not a scope: `struct` body

- (CPU) stack: More capable than Algo 1 stack data structure
  - New variable: *pushed* to end (usually low address)
  - After scope exit: *popped* from end
  - (Hence the name)
- Additionally: random access in stack frame (area of function)
- Recursion: One function has multiple stack frames, one for each recursive call

Heaps

Heap, Heap, I'm a Sheep

```
int *iota(int num, int start) {
    /* create values start, start + 1,
    start + 2, ..., start + num - 1*/
    int *values = // help!!!
```

What kind of memory do we want?

- Persists over function boundaries
- Can be "safely" shared across threads
- Type-independent
- Dynamically sized
- "Leak-free"
- Fast allocations

C solution:

```
void *malloc(size_t num_bytes);
void free(void *pointer);
```

Issues?

- **Alignment**: `char arr[sizeof(long long)];` and `long long` have same size, but different alignment
  - Solution: Assume worst-case alignment
- **Locality**: No option to allocate, e.g., list in adjacent memory
  - Solution: No real solution
- **Fragmentation**: No option to allocate memory with similar livetimes in similar memory
  - Solution: fancy heuristics in fast malloc implementations

`malloc()` Implementations

`free()`-styler!

DESIGN IMPLICATIONS

- `malloc()` needs to go into `free()`
  - Only information: the pointer
  - We need to track all sizes of allocated memory
  - Store somewhere (e.g., around allocated memory)
- Need to reuse memory: find previously `free()`d memory
  - Need a lookup structure
  - For fast implementations: acceleration structure
- Need to track whether new memory needs to be requested ⇒ OS!
  - Issue: Fragmented size does not count, only contiguous free memory
- Need to actually get memory from the OS...

I want to `brk()` free

`mmap()` men, `mmap()` men, `mmap()` `mmap()` `mmap()` men

GETTING MEMORY FROM THE (LINUX) OS

SBRK()

`sbrk()` sets "program break", the size of the shared data segment

- Can be used for small allocations
- Needs data structure that shows whether lowest data can be freed to shrink "break"
- Big performance issue: Extremely diverse size of allocations
  (1 - 128 kiB), similar sizes should be adjacent to combat fragmentation and to speed-up lookups.

MMAP()

`mmap()` maps files (and chunks of memory) into address space

- Can be used for all allocations
- Technical limitation: multiples of page size (usually 4 kiB)
- OS needs exact information on class="text-unimportant" `munmap()`
- Setup security features like "Guard Pages"
- Performance issue: demand paging and pre paging

`malloc()` Shenanigans

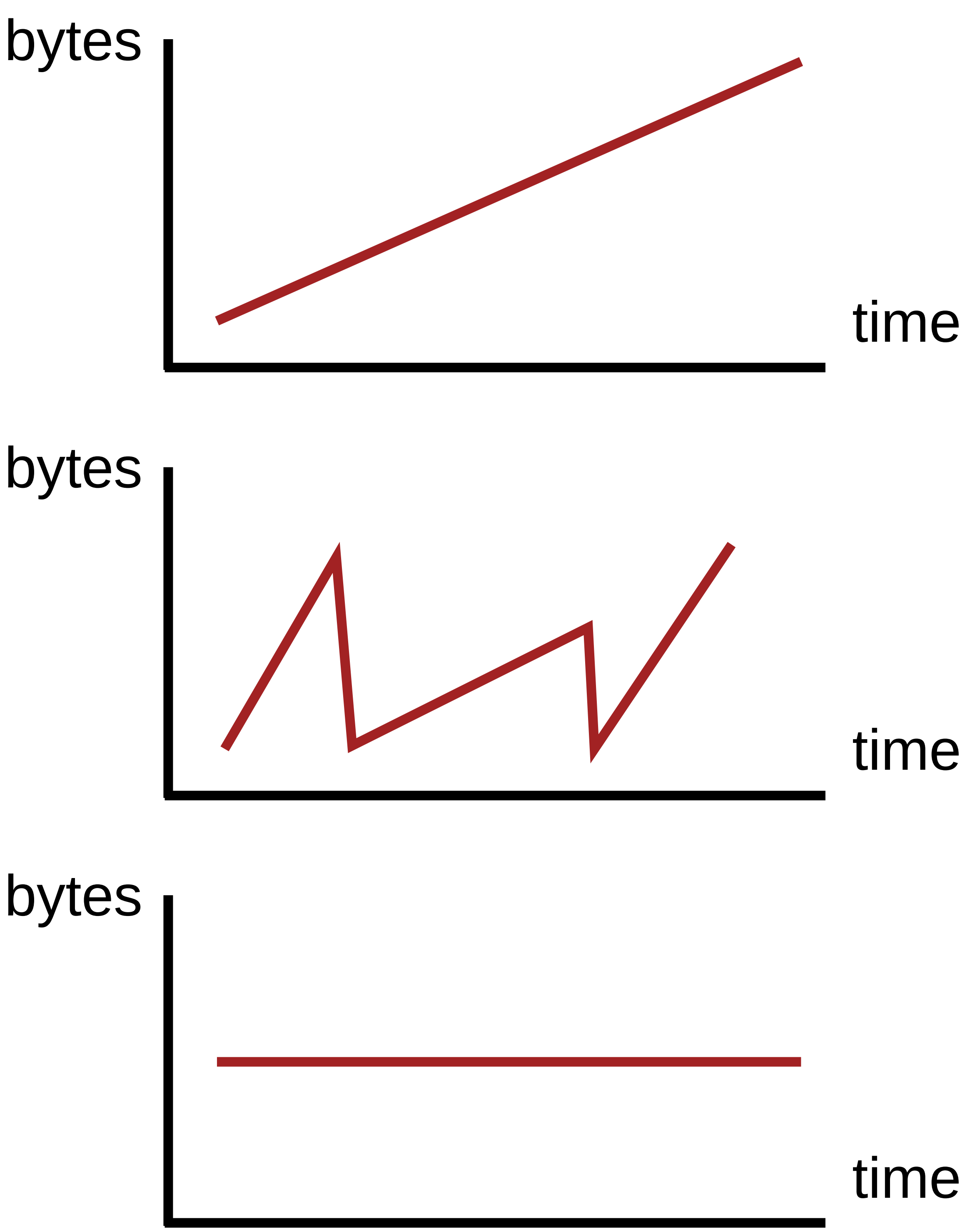
Embrace the Penguin

- Does `malloc()` fail? No (except in OS assignments). Still check!
- Why not? Demand Paging
- What happens on out of memory? Out of Memory (OoM) killer
- What is targeted by OoM killer? Applications with a lot of memory, e.g., the X-server, the desktop manager, or *your application...*
- What do you do then? End other applications or get more RAM

Allocation Strategies

`OS *copy = malloc(pasta)`: Allocation Patterns

- Ramp
  - Allocate more and more, does not free
  - Example: file system cache
- Peaks
  - Allocate in phases
  - Example: pipelined processes
- Plateau
  - Does not allocate, (only at beginning)
  - Example: long computations



Applications have some typical allocation pattern
adapted from source

Allocation Strategies

Please a Bit - Bitmap Allocator

- Pick a size: e.g., 64 byte (cache line!)
- Reserve a memory range for actual storage
- Reserve a memory space where you store one bit per byte
- Allocate by rounding up size, finding consecutive free block large enough
- Set bits for "allocated"
- On free? Need to find length
- Malloc-style allocator: size lookup somewhere else
- Other, great idea: use only for sizes ≤ blocksize, free becomes trivial

PERFORMANCE🚴

- Bit range: extremely hot ⇒ great
- size lookup for free() slow
- If single size (or range, e.g., 32-64 bytes): Extremely fast ⇒ *Slab Cache*
- Programs often allocate same size often

Allocation Strategies

Free List

- Linked List: [next, length]-segments
- Store in returned memory (no space overhead!)
- On alloc: iterate through list until length ≥ requested length
- Remove from list, return segment
- Possibly add new, smaller block to list
- Issue: alignment
- On free? Need to find length
- Simple idea: store before returned pointer (mind the extra space)
- Remember to merge segments if possible

PERFORMANCE🚴

- Implementation advantage: Simple to implement, ok-ish for all allocation sizes
- Performance issue: linked-list
- Expectation: Newly returned memory hot. Usually wrong, applications defer free()
- Remember? Programs often allocate same size often
- Usually, first try succeeds ⇒ actually decent!

Allocation Strategies

My Best Buddy - Buddy Allocator

- Allocatable memory size: $2^n, n \in \mathbb{N}$
- Allocation request: round up to next power of two
- Free-list of all blocks of required size. If empty: (recursively) allocate one power larger, split result and insert other block (= buddy) into free-list
- On free: insert now freed block into free-list. If buddy is also free, merge (recursively) to regain large blocks
- Issues:
  - Internal & external fragmentation
  - Space waste up to $\frac{1}{2} - \varepsilon$

PERFORMANCE🚴

- Possibly replace free-list with other tool.
- Advantage: Simple to implement
- Performance: binary split really fast
- Depending on free-list or replacement: merge slow(ish)
- Not (really) general purpose, but great building block!

Allocation Strategies

Welcome To the Arena

- Remember application phases? Wouldn't it be great to only cleanup once at the end of the stage?
- Idea: bin of memory (with parents)
- Allocate to bin that lives as long as needed, but not longer
- At the end of live: `deallocate_all(arena *)`
- Here: no real implementation advice, depends on your use case

PERFORMANCE🚴

- Hugely depends on correct usage, but then can be really fast
- No automatic allocation to correct arena possible, needs programmer input
- But does it? Machine learning to the rescue!
  - Learning-based Memory Allocation for C++ Server Workloads
  - Combining Machine Learning and Lifetime-Based Resource Management for Memory Allocation and Beyond
  - Performance Test Selection Using Machine Learning and a Study of Binning Effect in Memory Allocators

Allocation Strategies

Composition is Key

- No allocator great alone
- But together: great building blocks
- General Pattern: Different behavior based on size (and alignment)

Example

- If size $\geq$ 2MB $\Rightarrow$ call `mmap()` directly
- If 2MB $>$ size $\geq$ 128kB $\Rightarrow$ Use page-aligned free-list backed by `mmap()`
- All below: backed by buddy allocator, all buddy-managed memory initially 2MB aligned
- If 128kB $>$ size $\geq$ 4kB $\Rightarrow$ Use different page free-list
- If 4kB $>$ size $>$ 128 B $\Rightarrow$ Use 64 B-aligned free-list
- If 128 B $\geq$ size $>$ 64 B $\Rightarrow$ Use 128 B slab cache
- If 64 B $\geq$ size $>$ 48 B $\Rightarrow$ Use 64 B slab cache
- If 48 B $\geq$ size $>$ 32 B $\Rightarrow$ Use 48 B slab cache
- If 32 B $\geq$ size $>$ 16 B $\Rightarrow$ Use 32 B slab cache
- If 16 B $\geq$ size $\Rightarrow$ Use 16 B slab cache

union

United We Stand, Devided We Undefined Behavior (UB)

```
union foo {
    long long lvalue;
    double dvalue;
};
```

- Either store lvalue or dvalue
- Can NOT be used to convert (at least not savely)
- Only read from object last written to!
- Basically only useful for saving memory
- And if we want to convert?
- use `memcpy()`
- Performance🚴: Pretty quick, but use with caution due to UB

Recap

What did we learn today?

- What is the branch predictor?
- What is the preprocessor?
- `malloc` and `free()`
- Where and how do we allocate memory?
- What is a `union`?

And next lecture? Not a lecture, first practical session

- Practical introduction
- Repository creation
- Exercise introduction