Lab 3: Memory allocation

Hung-Wei Tseng
The PA of Lab 3 is optional

• So you have time to prepare for CSE142 midterm
• You only need to do it if you want an A+
• You can turn it in any time before 9/2
• You will have to read through everything and figure out how it works yourself — a challenge for getting an A+!
• You cannot use it to replace any other Lab’s PAs.
  • e.g., If you miss Lab 5 PA, but submit Lab 3 PA — you’re still getting an A- for 142L
#include <cstdint>
#include <vector>
#include <algorithm>
#include "function_map.hpp"

struct MM {
    struct MM * next; // I know that pointers are 8 bytes on this machine.
    uint64_t junk[7]; // This forces the struct MM to take up a whole cache line,
    abolishing spatial locality.
};

extern "C"
struct MM * miss(struct MM * start, uint64_t count) {
    for(uint64_t i = 0; i < count; i++) { // Here's the loop that does this misses. It's
        start = start->next;
    }
    return start;
}

extern "C"
uint64_t * miss_machine(uint64_t * data, uint64_t size, uint64_t arg1) {
    #define ARRAY_SIZE (8*1024)
    auto array = new struct MM[8*1024]; // This is bigger than the L1

    // This is clever part 'index' is going to determine where the pointers go. We fill
    it consecutive integers.
    std::vector<uint64_t> index;
    for(uint64_t i = 0; i < ARRAY_SIZE; i++) {
        index.push_back(i);
    }
    // Randomize the list of indexes.
    std::random_shuffle(index.begin(), index.end());

    // Convert the indexes into pointers.
    for(uint64_t i = 0; i < ARRAY_SIZE; i++) {
        array[index[i]].next = &array[index[(i + 1) % ARRAY_SIZE]];
    }
    MM * start = &array[0];
    start = miss(start, 1024*1024*128); // 128 million accesses.
    return reinterpret_cast<uint64_t*>(start); // This is a garbage value, but if we
do not return it, the compiler will optimize out the call to miss.
}
What is the “miss machine” doing?
Outline

• Why “miss machine” is bad
• How fast is my memory?
• How big is my data?
• How memory stores my data?
Recap: Memory Hierarchy

Processor

Core

Registers

SRAM $

DRAM

L1 $

L2 $

L3 $

Storage

larger

fastest

< 1ns

a few ns

tens of ns

us/ms

TBs
Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then PC + 4
  - Data — the current element in an array, then the next

- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
  - Data — the same data can be read/write many times

If we want to improve the memory performance of our code, we have to make our code better exploit “localities”
Performance evaluation with multi-level $\$ $

![Diagram showing memory hierarchy and cache hit rates.]

- Processor Core
  - Registers
  - L1 $\$: 90% hit rate, no penalty
  - L2 $\$: 10% hit rate, 10 cycles
  - DRAM: 60% hit rate, 52 cycles

ld 0xDEADBEEF

Every instruction fetch access memory

Assume 30% data accesses

\[
1 \times (1 + 30\%) \times (1 - 90\%) \times [10 + (1 - 60\%) \times 52] = 5 \text{ cycles}
\]
How fast is my memory hierarchy?
The tool

• Memory Latency

• Spec
  • dmidecode — for DRAM parameters
  • lscpu — for on-CPU caches
Memory is Slow

- Our machine: 3500MHz => 0.28ns/cycle
- L1 latency is 4 cycles -> L1 hit penalty = 3 cycles.
- L2 latency is 23 cycles -> MissPenalty = 19 cycles
- L3 latency is 96 cycles -> MissPenalty = 92 cycles
- DRAM latency is 217 cycles! -> MissPenalty = 213 cycles
Memory performance when running applications
Misses Per Instruction

• In the lab we will use Misses-per-Instruction (MPI)
  • It’s more useful than miss rate for measuring application performance
  • We can’t easily measure miss rate on our processors
• $ET = IC \times CPI \times CT$
• $CPI_{tot} = CPI + MPI \times MissPenalty$
Consequences of Slow Memory: the PE

- A good CPI in Labs 1 and 2: ~0.5 cycles.
- What MPI would reduce performance by 1/2?
  - i.e., increase CPI by 0.5
- L1_MPI
  - $0.5/29 = 0.017$ -- 17 L1 misses per 1000 instructions
- L2_MPI
  - $0.5/92 = 0.005$ – 5 L2 misses per 1000 instructions
- L3_MPI
  - $0.5/213 = 0.002$ – 2 L3 misses per 1000 instructions
- L1_HPI (hits per instruction or just loads/instruction)
  - $0.5/3 = 0.166$ – 17% of instructions
How big is a struct?
Consider the following data structure:

```c
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What’s the output of

```c
printf("%lu\n", sizeof(struct student));
```

A. 20  
B. 28  
C. 32  
D. 36  
E. 40
Consider the following data structure:

```c
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What's the output of

```c
printf("%lu\n", sizeof(struct student));
```

A. 20  
B. 28  
C. 32  
D. 36  
E. 40
The result of sizeof(struct student)

- Consider the following data structure:
  ```c
  struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
  };
  ```
  What’s the output of
  `printf("%lu\n", sizeof(struct student));`?
  
  A. 20  
  B. 28  
  C. 32  
  D. 36  
  E. 40
Memory addressing/alignment

• Almost every popular ISA architecture uses “byte-addressing” to access memory locations
• Instructions generally work faster when the given memory address is aligned
  • Aligned — if an instruction accesses an object of size $n$ at address $X$, the access is \textit{aligned} if $X \mod n = 0$.
  • Potentially incurs two cache misses for an access
  • Some architecture/processor does not support aligned access at all
  • Therefore, compilers only allocate objects on “aligned” address
Memory addressing/alignment

- Unaligned accesses are sometimes inefficient: one load can cause 2 cache misses
- I haven’t been able to demonstrate this on our machines
  - No effect for L1.
  - Measurement noise for L2/L3/DRAM is too large
- Some versions of ARM’s ISA have very complicated semantics for unaligned accesses
- Other ISAs
  - Unaligned access can cause an interrupt.
How “Tensors” are laid out?
Tensors

- A tensor is an algebraic object that describes a multilinear relationship between sets of algebraic objects related to a vector space.
- In short, an N-dimensional object with some mathematical meaning.
Tensor Layout

- Internally, tensors are stored as a linear array
  - size = x*y*z*b
- Here's the code to translate coordinates to a linear index (tensor_t.hpp)
  - Increment x moves the index by 1
  - Incrementing y moves by size.x
  - Incrementing z moves by size.x*size.y
  - Incrementing b moves by size.x*size.y*size.z

\[\text{index} = b \times (\text{size.x} \times \text{size.y} \times \text{size.z}) + z \times (\text{size.x} \times \text{size.y}) + y \times \text{size.x} + x;\]
Moneta Tour

- Figuring out which tag is which
- Zooming to tag
- Resetting zoom
- Locking axes
- How layers work.
The programming assignment (optional)
You will implement a fixed-sized, alignment-aware, locality-optimizing memory allocator

- Memory allocators are really important
- Even google releases their own memory allocator — https://github.com/google/tcmalloc
  - Malloc()/free in C, new/delete in C++

General purpose allocators can allocate/deallocate memory of any size.
  - This is requires a lot of complexity (we’ll discuss this in more detail next week)
  - And they can be a little slow, if memory allocation is important to your program
  - Many systems allocate many, many small objects (often of a single type) very fast.

Optimize for the common case
  - Build an allocator for just for that type.
  - Simpler, faster.
# The “reference allocator”

```c++
#include <stdlib.h>

template<
    class T,
    size_t ALIGNMENT
    >

class ReferenceAllocator {
    std::set<T*> chunks; // We store everything we allocated so we can clean up in the destructor.

    public:
    typedef T ItemType; // This will make T available as ReferenceAllocator::ItemType
    static const size_t Alignment = ALIGNMENT; // Likewise, we can access the alignment as ReferenceAllocator::Alignment

    ReferenceAllocator() {}  

    T * alloc() { 
        void* p = NULL; 
        // this system call can allocate arbitrary-sized and aligned 
        // objects. Since it can handle any size, it's more general. 
        int r = posix_memalign(&p, ALIGNMENT, sizeof(T));
        if (r == -1) { 
            std::cerr << "posix_memalign() failed. Exiting: " << strerror(errno) << "\n";
            exit(1);
        }
        uint8_t * t = reinterpret_cast<uint8_t*>(p);
        // alloc_chunk provides void*, but we can assign to void. So cast...
        for(uint i = 0; i < sizeof(T); i++) {
            t[i] = 0; // and set to zero.
        }
        T* c = reinterpret_cast<T*>(p); // cast to the type we allocate.
        new (c) T; // This is the "in place" new operator. It constructs an object at a given location.
        chunks.insert(c); // record it so we can delete it later.
        return c;
    }

    void free(T * p) { 
        std::free(reinterpret_cast<void*>(p)); // Return the memory
        chunks.erase(p); // note that it's no longer allocated.
    }

    ~ReferenceAllocator() { 
        for(auto & p: chunks) { 
            std::free(reinterpret_cast<void*>(p)); // Return everything that still allocated.
        }
    }
};
```
"Your" allocator

```cpp
#include <stdlib.h>
#include <iostream>
#include <string.h>
#include "ChunkAlloc.hpp"
#include <sys/mman.h>
#include "pin_tags.h"

static size_t allocated_chunks = 0;

void init_chunk() {
    // This creates an tag that covers no memory, since the max is very small and min is very large.
    // We'll grow it below.
    TAG_START("chunks", reinterpret_cast<void*>(-1), reinterpret_cast<void*>(0), true);
}

void * alloc_chunk() {
    // allocate CHUNK_SIZE bytes of memory by asking the operating system for it.
    // this is actually malloc gets it's memory from the kernel.
    // mmap() can do many things. In this case, it just asks the kernel to
    // give us some pages of memory. They are guaranteed to contain zeros.
    void * r = mmap(NULL, CHUNK_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, 0, 0);
    if (r == MAP_FAILED) {
        std::cerr << "alloc_chunk() failed. This often means you've allocated too many chunks. Exiting: " << strerror(errno) << "\n";
        exit(1);
    }
    TAG_GROW("chunks", r, reinterpret_cast<uint8_t*>(r) + CHUNK_SIZE);
    allocated_chunks++;
    return r;
}

void free_chunk(void *p) {
    // Return the chunk to the OS. After this, accesses to the addresses in the chunk will result in SEGFAULT
    int r = munmap(p, CHUNK_SIZE);
    if (r != 0) {
        std::cerr << "free_chunk() failed. exiting: " << strerror(errno) << "\n";
        exit(1);
    }
    allocated_chunks--;
}

size_t get_allocated_chunks() {
    return allocated_chunks;
}
```
You're going to build your own version of `ReferenceAllocator` called `SolutionAllocator`. You'll find a copy of `ReferenceAllocator.hpp` in `SolutionAllocator.hpp`. Do your work there.

- All addresses that `SolutionAllocator::alloc()` returns must be aligned to `ALIGNMENT` so `addr % ALIGNMENT == 0`.
- All addresses that `SolutionAllocator::alloc()` returns must point to at least `sizeof(T)` bytes of memory.
- `SolutionAllocator::alloc()` needs to set all bytes of memory that the instance of `T` will occupy to zero before constructing `T`.
- `SolutionAllocator::alloc()` needs to construct an instance of `T` in the memory using the in-place `new` operator (see below).
- `SolutionAllocator::alloc()` cannot return the same pointer twice unless the pointer has been deallocated with `SolutionAllocator::free()` first.
- After the destructor completes, `SolutionAllocator` must have called `free_chunk()` for every chunk it allocated with `alloc_chunk()`. I.e., `get_allocated_chunks()` must return 0.
- You are free to use the STL data structures for the internals of `SolutionAllocator`, but `alloc()` may not return any memory that is a part of an STL container.
- The only mechanism you can use to allocate memory is `alloc_chunk()/free_chunk()`. No calls to `malloc()` (or other functions from the standard library that allocate raw memory) or `new` (other than the “in place” version.) to allocate the space `SolutionAllocator` will return.
- Your allocator must recycle: If memory is returned to it via `free()`, your allocator should reallocate that memory before requesting new memory via `alloc_chunk()`. This prevents your allocator from continually allocating new memory, which is very fast, but not a realistic solution.

You're going to build your own version of `ReferenceAllocator` called `SolutionAllocator`. You'll find a copy of `ReferenceAllocator.hpp` in `SolutionAllocator.hpp`. Do your work there.

- All addresses that `SolutionAllocator::alloc()` returns must be aligned to `ALIGNMENT` so `addr % ALIGNMENT == 0`.
- All addresses that `SolutionAllocator::alloc()` returns must point to at least `sizeof(T)` bytes of memory.
- `SolutionAllocator::alloc()` needs to set all bytes of memory that the instance of `T` will occupy to zero before constructing `T`.
- `SolutionAllocator::alloc()` needs to construct an instance of `T` in the memory using the in-place `new` operator (see below).
- `SolutionAllocator::alloc()` cannot return the same pointer twice unless the pointer has been deallocated with `SolutionAllocator::free()` first.
- After the destructor completes, `SolutionAllocator` must have called `free_chunk()` for every chunk it allocated with `alloc_chunk()`. I.e., `get_allocated_chunks()` must return 0.
- You are free to use the STL data structures for the internals of `SolutionAllocator`, but `alloc()` may not return any memory that is a part of an STL container.
- The only mechanism you can use to allocate memory is `alloc_chunk()/free_chunk()`. No calls to `malloc()` (or other functions from the standard library that allocate raw memory) or `new` (other than the “in place” version.) to allocate the space `SolutionAllocator` will return.
- Your allocator must recycle: If memory is returned to it via `free()`, your allocator should reallocate that memory before requesting new memory via `alloc_chunk()`. This prevents your allocator from continually allocating new memory, which is very fast, but not a realistic solution.

`SolutionAllocator` already satisfies all of these except the last two: `ReferenceAllocator` uses `posix_memalign()` which is forbidden in your solution. With respect to recycling, I'm not sure how `posix_memalign()` works internally, so I'm not sure how it recycles, but it does something reasonably efficient. You'll find that removing `posix_memalign()` and meeting the above criteria will require you to rewrite most of the code in `SolutionAllocator.hpp`. 
Here are the basic steps that your memory allocator must accomplish over its lifetime.

1. Initialize itself and its internal data structures.
2. Respond to calls to `alloc()`
   - If the allocator has recycled objects, initialize one and return it.
   - Otherwise, if the allocator has new (not recycled) memory on hand, initialize one object worth, and return it.
3. Otherwise, if the allocator has no memory on hand, call `alloc_chunk()`, and goto 2.2.
4. Respond to calls to `free()`
   - Store the object somewhere so it can be recycled.
5. On destruction, called `free_chunk()` to deallocate all the memory you allocated with `alloc_chunk()`. 

The Lifecycle of a Memory Allocator
Main Challenges

• Acquiring a large chunk of memory and dividing it into properly-constructed C++ objects.
• Ensuring the alignment of those objects is correct.
• Deciding which object to return for a given allocation.
• Freeing and recycling objects efficiently.
• Ensuring that all the chunks you allocate are freed by the destructor.
• Maximizing locality among allocated objects.
Some ideas...

- The main difference between what you'll implement in `SolutionAllocator` and `posix_memalign()` is that `SolutionAllocator` only needs to allocate objects of a single size. You can exploit this fact to improve performance.

- `alloc_chunk()` lets you allocate enough space to store many objects. Since the objects are all the same size and alignment, you can calculate where each of instance of `T` will reside with the chunk.

- You need to recycle. So, think about how you can efficiently store `free()`-ed memory while it's waiting to be re-`alloc()`-ed.

- You will need to choose the right data structures and algorithms to achieve good performance. Think about what each data structure _needs_ to do and what operations are most important to performance. Use the STL! (In Lab 2, I noticed many students implementing things that the STL already provides. Don't re-invent -- or debug -- the wheel!)
Reference Implementation

• Not a valid solution
  • Doesn’t use alloc_chunk()
• Demonstrates
  • placement new
  • reinterpret_cast<>()
Steps

• With that in mind, start with the simplest ones and go from there. I'd proceed in this order:
• The `microbench` function.
• The `bench` function.
• The `miss_machine` function.
Evaluation

- 8 benchmarks total (Allocator.cpp)
  - Don’t worry – performance is highly correlated.
- 4 microbenchmarks
  - Small object allocation
  - Small object deallocation
  - Large object allocation
  - Large object deallocation
- 3 Benchmarks
  - Mixed alloc/free of small objects
  - Mixed alloc/free of medium objects
  - Mixed alloc/free of large objects
- 1 Locality-aware benchmark
  - A miss machine.
Micro Benchmarks

• Allocate many times, measure latency
• Deallocate many times, measure latency.
Benchmarks: exercise()

- Simulates an “interesting” program
  - Allocate a bunch of objects
  - Deallocate some at random
  - Allocate replacements
- Some objects will live a long time, some will die young.
Benchmark: Miss Machine

- MissingLink is 8 Bytes
  - 4096 of them should fit in our 32kB cache
  - 512 64-byte cache lines
- But can your allocator make them fit?
How To Succeed on This Lab

• Start early
  • It was possible to do pretty well on the Lab 2 PA starting 12 hours before the deadline.
  • It is not possible for the Lab 3

• PA (Optional, do it only if you want an A+)
  • Start simple
  • Try some of the tools

• Come to office hours!