1 Arrays

1.1 1-dimensional arrays

A 1-dimensional array is a contiguous block of memory capable of storing n elements of a given type.

```
char msg[12];
char *msg = malloc(12*sizeof(char));
char msg[12] = {'H', 'i', '!', ...};
```

Let `%rdi` be `T *arr` and `%rsi` hold the offset.

```
get_digit:
  movl (%rdi, %rsi, 4), %eax
```

1.2 n-dimensional arrays

A n-dimensional array provides a row-major ordering, such that each *array* is laid out contiguously in memory. `get_digit` is:

```
get_digit:
  leaq (%rsi, $rdx, 1), %rax
  addq %rax, %rdx
  movl %rdi(, %rsi, 4), %eax
```

1.3 Multi-level arrays

Multi-level arrays store sub-arrays as pointers in the top-most array, requiring several memory accesses:

```
get_digit:
  salq 2, %rsi
  movl %rdi(, %rdi, 8), %rsi
  movl (%rsi), %eax
```

2 Structures

A structure allows us to store many different types together in memory. Access field `m` of type τ (instance `r`) by `r.m`, and `r->m` (when `r` : `*`τ).

Recall that each member of a structure of type τ (members denoted τ_i) must be aligned along $\alpha \cdot |\tau_i|$. τ itself must be aligned along $\max_{i \leq |\tau|} |\tau_i|$

2.1 Unions

A union stores fields as does a structure, but fields are laid "on top" of each other in memory. $|\mu| = \max_{i \leq |\mu|} \text{width}(\mu_i)$.

3 Caches

Caching is effective because of two localities.

Temporal locality Temporal locality refers to the property that memory that is accessed is likely to be referenced again in the future.

Spacial locality Spacial locality refers to the property that memory will be read at similar locations in time.

The caching performance is defined as AMAT $=$ HT $+$ MR $\cdot$ MP.

When determining the value of memory returned from a cache lookup, find first the index of the set. Then find the matching tag (or fault), ensure that it is valid (or fault). Find the offset, and return that location.

(T)	(I) $s = \log_2(C/K/E)$	(O) $k = \log_1(K)$

3.1 Block Size (K)

A block is a single unit of transfer. A set can contain many blocks, and the offset within a block is $k = \log_2(K)$.

3.2 Cache Size (C)

C is the total size of a cache. The total number of sets is C/K, and the set that a block belongs to is given as $s = \log_2(C/K/E)$

3.3 Associativity (E)

E refers to the number of refers to the number of blocks in a set.

3.4 Types of Cache Misses

There are three types of cache misses:

Compulsory miss Always occurs on the first access in a block.

Conflict miss Occurs when multiple locations map to the same set or block.

Capacity miss Occurs when the set of active blocks is larger than the cache.

3.5 Cache writing

On write-hits, we can either *write-through* and immediately write to the next level, or *write-back* and write on eviction. On write-misses we can either *write-allocate* and load into cache, or *no-write-allocate* and write immediately to memory.

4 Control Flow

There exist both synchronous and asynchronous exceptions, which define "exceptional control flow". In this model, the CPU (running a program in user-land) receives an exception, and transfers control to the kernel, which then returns control either to the current or next instruction, *or* aborts the program.

Traps System calls, breakpoint traps, etc.

Faults Unintentional and (semi-)recoverable transfers of control. Paged out memory, divide-by-zero, segmentation faults, etc.

Aborts Unintentional and unrecoverable.

The CPU "juggles" more processors than (virtual) cores. To do so, it *context switches* between running processes, by (1) saving the current registers to memory, (2) scheduling the next process for execution, and (3) loading the saved registers and switching address space.

To switch the running process, we may `fork()` and `exec()`.

5 Virtual Memory

Let MMU : VA $\mapsto$ PA. Divide the virtual and physical address as follows:

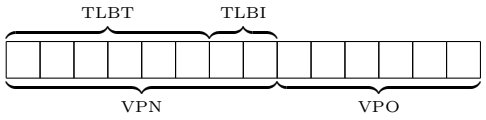

Figure 4.1: Layout of 14-bit virtual memory.

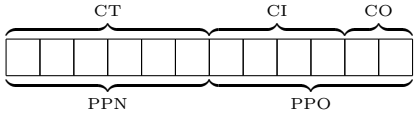

Figure 4.2: Layout of 12-bit physical memory.

1. Determine the TLBI and scan for the PPN corresponding to a valid TLBT.
   (a) If one is found, copy to the physical address.
   (b) If not, read the entire VPN from the VA and ensure that it's valid. If one exists, copy its PPN to the PA. If not, page fault and load the entry from disk (update TLB).
2. Determine the CT, CI, CO. Find the corresponding index, then a matching tag, and check the block is valid.

Let $|\text{VPO}| = \log_2(P)$ (where P is the size of a single page) be the low-order bits of a VA. Let |VPN| be the remaining high-order bits in the virtual address.

Let $|\text{TLBI}| = \log_2(\#_s)$ be the high-order bits of the VPN, where $\#_s$ is the number of sets in TLB. The remaining bits are the TLBT.

The low-order bits of physical memory PPO $\cong$ VPO in all cases. Let the remaining high-order bits be the PPN.

6 Memory Allocation

There are two classifications we use to disambiguate memory allocators:

Explicit allocator(s) The programmer must allocate and free memory manually.

Implicit allocator(s) The programmer need only allocate memory, freeing is done automatically.

We wish to maximize "throughput" over R_i while maximizing $U_k = \frac{1}{H_k} \cdot \max_{i \leq k} P_i$ (where H_k is the current size of the heap and P_i is the payload of the i-th request).

Fragmentation is the antithesis of this goal. Internal fragmentation refers to unused space in a block, and block or structure padding. External fragmentation refers to "holes" between blocks.

6.1 Memory Allocation in C

We detail the pertinent functions from `<stdlib.h>`:

```
void *malloc(size_t size);
void *calloc(size_t nr, size_t size);
void *realloc(void *p, size_t size);
```

And those pertaining to reclaiming memory:

```
void free(void *p);
```

`malloc()`, `calloc()` and `realloc()` all allocate memory. `malloc()` allocates a number of aligned, contiguous bytes, though successive allocations may not be contiguous. `calloc()` performs the same operation, also zero-ing out memory, and has a different signature. `realloc()` changes the size of an existing block of memory.

6.2 Free-list architecture

Implicit free list Let all blocks contain the size and tag bits duplicated at the header and footer of a block.

First-fit Traverse starting at the root until a free block is seen.

Next-fit First-fit from the last allocated block.

Best-fit Traverse the whole heap, H and minimize extra padding.

To free blocks, unset the allocated bit and coalesce (recalling that headers are duplicated).

Explicit free list Store only the size and tags header for allocated blocks, and store only the size, tags, and the locations of the next and previous free block in free blocks. Traverse over only free blocks in any above algorithm. Can be a linear or logically/physically-circular linked list.

6.3 Garbage collection

Let the heap be treated as a *graph*, such that each node in V is an allocated block of memory. Let each edge $p \in P \subseteq V \times V$ be a pointer to another node within V.

To preform garbage collection over a graph G, traverse G in any total-ordering, beginning at the collection of root nodes. For any non-marked node, mark it and traverse over its children. To remove non-reachable blocks, traverse the graph again, un-marking marked blocks, and freeing non-marked blocks.

7 Java

Primitives are stored in mostly the same fashion as they are in C in Java. A notable exception is that: (1) `chars` in Java are 2-byte UTF-8 code-points, whereas they are 1-byte ASCII in C.

Note that arrays in Java contain an additional length field that is bounds checked. Note also that arrays in Java are stored on the heap (in all cases), whereas they can be stored on the stack in C.

Strings in Java are not bounded by a `NUL` character as they are in `C` and are read-only.

Note that Java guarantees that fields will be zero-initialized by a constructor.

7.1 vtable

Each instance of an object in Java contains a header and a vtable. The header contains information pertinent to garbage collection, hashing, and etc. The vtable is thought of as a jump table for virtual methods and other class information. There is only one vtable per-class, so each instance has a pointer to it.

When a class is extended (such that $\tau' \prec \tau$), the fields in τ' are stored *after* the fields in τ in the vtable. Therefore, it is safe to *use any sub-type of τ where a τ is expected, satisfying covariant method arguments.*