CSE351 A, Grossman, Final Exam

Taylor Blau[1]

Spring, 2018

[1]Material is compiled from the lecture notes, the readings in *Computer Systems* (3rd edition), Bryant, O'Hallaron, and my own notes.

Contents

1 Arrays **5**
- 1.1 Allocation . . . 5
  - 1.1.1 Allocation . . . 5
  - 1.1.2 Initialization . . . 5
- 1.2 1-dimensional arrays . . . 5
  - 1.2.1 Indexing . . . 5
  - 1.2.2 Guarantees . . . 6
- 1.3 n-dimensional arrays . . . 6
- 1.4 n-level arrays . . . 6

2 Structures & Unions **9**
- 2.1 Structures . . . 9
  - 2.1.1 Structure Members . . . 9
  - 2.1.2 Structure Representation . . . 9
  - 2.1.3 Alignment Principles . . . 10
  - 2.1.4 Internal alignment . . . 10
  - 2.1.5 External alignment . . . 10
- 2.2 Unions . . . 11

3 Caches **13**
- 3.1 Cache Organization . . . 13
  - 3.1.1 TIO-breakdown . . . 14
  - 3.1.2 Associativity . . . 15
  - 3.1.3 Eviction Policies . . . 15
  - 3.1.4 Types of Cache Misses . . . 16
  - 3.1.5 Cache Writes . . . 16

4 Control Flow, Processes **17**
- 4.1 Motivation . . . 17
- 4.2 Processes . . . 17
  - 4.2.1 Running Exceptions . . . 17
  - 4.2.2 Exception Table . . . 17
- 4.3 Asynchronous Exceptions (Interrupts) . . . 17
- 4.4 Synchronous Exceptions . . . 18
- 4.5 Processes . . . 18

4.5.1 Context Switch(-ing) . . . 18
4.6 Creating Processes . . . 19
4.6.1 `fork()` . . . 19
4.6.2 `exec*()` . . . 19

5 Virtual Memory **21**
5.1 Motivation . . . 21
5.2 Address Translation . . . 21
5.2.1 Physical Page Lookup . . . 22
5.2.2 Page Faults . . . 22
5.2.3 Overview . . . 22
5.2.4 Basic Parameters . . . 22
5.2.5 Components of virtual address (VA) . . . 23
5.2.6 Components of physical address (PA) . . . 23

6 Memory Allocation **25**
6.1 Memory Allocation in C . . . 25
6.1.1 `malloc()` . . . 25
6.1.2 `free()` . . . 26
6.2 Performance Goals . . . 26
6.2.1 Throughput . . . 26
6.2.2 Peak Memory Utilization . . . 26
6.3 Fragmentation . . . 26
6.3.1 Internal Fragmentation . . . 26
6.3.2 External Fragmentation . . . 26
6.4 Implicit free list . . . 27
6.4.1 First fit . . . 27
6.4.2 Next fit . . . 27
6.4.3 Best fit . . . 27
6.4.4 Allocating a Free Block . . . 28
6.4.5 Freeing an Allocated Bloc . . . 28
6.5 Explicit free list . . . 29
6.5.1 Linkages . . . 29
6.6 Segregated free list . . . 29
6.7 Garbage Collection . . . 30
6.7.1 Memory as a Graph . . . 30
6.7.2 Overview . . . 30
6.7.3 `mark()` . . . 30
6.7.4 `sweep()` . . . 31

7 Java **33**
7.1 Data in Java . . . 33
7.1.1 Primitives . . . 33
7.1.2 Arrays . . . 33
7.1.3 Strings . . . 33
7.1.4 Objects . . . 34

7.2 Methods & Method Dispatch 34
7.2.1 Constructors 34
7.2.2 Methods 34
7.2.3 Subclassing 35

Chapter 1

Arrays

It is often useful to store more than one element of a particular type T when programming in C. To solve this problem, we can use arrays, which are particularly different from arrays in other contemporary languages in a few ways which we detail below.

1.1 Allocation

1.1.1 Allocation

To allocate n elements of type T, write the following:

```
char msg[12];
char *msg = malloc(12*sizeof(char));
```

Note that in the first call, we declare `msg` to be an identifier holding the address of the first `char` in memory against a list of 11 more characters, comprising twelve in total. In the call involving `malloc()`, we make this explicit: we declare 12 times the size of the size of a single character (`sizeof(char)`).

To make this point more explicit the type of the expression `char[12]` *is* `char *`.

1.1.2 Initialization

One may also initialize elements within array while declaring that array, by writing the following or similar:

```
int xs[4] = {0, 1, 2, 3};
```

1.2 1-dimensional arrays

1.2.1 Indexing

Note that every element in an array of type T is *aligned*, such that each element, T_i starts at a location in memory that is a constant multiple of the size of T, i.e., that T_i resides at $\alpha \cdot$ `sizeof`(T), for some $\alpha \in \mathbb{N}$.

Consider this translated to x86-64. Recall the syntax $D(R_b, R_i, S)$, where D denotes displacement, R_b the base index, R_i the relative element index, and S the scale of that index.

Let's implement array indexing in assembly taking this approach:

```
int get_digit(int *arr, int n) {
  return arr[n];
}
```

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

Where `%rdi` contains `arr`, `%rsi` contains `n`, and `%eax` is the x86-64 calling convention-dictated return value register.

1.2.2 Guarantees

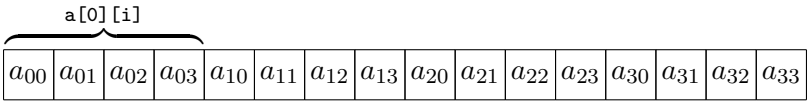
Note that C does not include a bounds checked array indexing operator, whereas Java does (and throws a `ArrayIndexOutOfBoundsException` if a bounds violation were to occur).

Therefore, C permits callers to do things like `arr[-1]` and `arr[`$\pm\infty$`]` (where ∞ is simply a large integer outside the bounds of `arr`), and simply returns whatever value is in that location in memory.

Note further that were multiple arrays to be located next to each other in memory (by the contract of `malloc()`, or otherwise), that an out-of-bounds array index from one array may simply return a value in the adjacent array. The simple statement of this all boils down to the fact that `all` memory lookups are scaled translations from a starting index, and C will happily return to you whatever is in memory at that location, or die with a SIGSEGV if this isn't possible.

1.3 n-dimensional arrays

C provides additionally a "row-major" ordering of n-dimensional arrays, illustrated as follows:



(where a_{ij} denotes the array lookup `a[i][j]`).

```
int get_digit(int **arr, int i, int j) {
  return arr[i][j];
}
```

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

Where we first indicate the starting index of the correct sub-array in `%rax`, then add the appropriate amount to that array, and then move it into the return location.

1.4 n-level arrays

C also permits us to treat sub-arrays as pointers into other arrays, as is the case with Java's implementation in the JVM. Whereas the code for `get_digit()` looks the same in either case, we no longer are guaranteed out-of-bounds behavior, or contiguity.

Consider, for example, the assembly instructions for this alternate version:

```
get_digit:
  salq $2, %rsi                # Shift %rsi to scale by sizeof(int)
  addq %rdi(, %rdi, $8), %rsi # First index into sub-array, then add
  movl (%rsi), %eax            # Then access sub-array, and move
  retq                         # Finally, return
```

Chapter 2

Structures & Unions

2.1 Structures

Often times we wish to define non-unary, structure data in C. To do so, C provides us with utility of structures–colloquially referred to as "structs"–in order to organize sequences of data with multiple types in regions of memory. Consider the following example:

```
struct {
  int a;
  int b;
} T;
typedef struct T T_t;
```

2.1.1 Structure Members

We can access structure members or fields in one of two cases.

Structure Instance Given a structure instance, we can access a field named m of a structure called r by writing `r.m`.

Structure Pointer Given a pointer to a structure, we can either write `(*r).m`, or use the convenience operator that C provides, which is semantically equivalent: `r->m`.

2.1.2 Structure Representation

Structures are represented as continuously allocated regions in memory, and we can refer to members within that structure by their name, as we have seen previously. Notably, each member (may) be of different type(s), so the question presents itself: how large is the structure? How do we know where each member is within that structure?

To answer this, C mandates that structures must be *aligned*, i.e., that each member of the structure must be itself aligned, and the entire structure must be aligned when laid out in an array.

Explicitly, C compilers make the following guarentees:

- A structure is represented as a block of memory, large enough to hold all of its fields.
- The fields within that structure are ordered in declaration order.

- The compiler determines the size and positions of the fields.

2.1.3 Alignment Principles

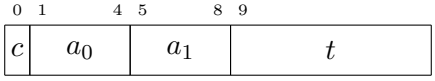
To review alignment principles, we recall that for each primitive data type of K bytes in width, that addresses must be a multiple of K, $\alpha \cdot K$. The motivation for such a recommendation comes from chip manufacturers, who promise that aligned memory accesses will be faster than un-aligned ones, mostly for caching purposes.

2.1.4 Internal alignment

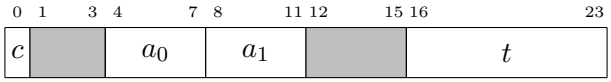
Consider the following structure:

```
struct {
  char c;
  int a[2];
  struct T *t;
} T;
```

Unaligned, the structure looks like the following:



But, this presents problems: note that both `a[0]`, `a[1]`, and `t` are all unaligned. To fix this, we add internal padding (otherwise referred to as internal fragmentation) in order to re-align the indices of a and t:



Note that we have added internal padding in order to satisfy the alignment constraints for the members of this structure.

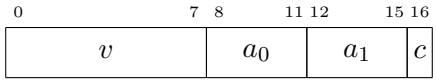
2.1.5 External alignment

Note that the overall structure must be aligned to $K_{\max}$, the alignment constraint of the largest sub-structure. In other words, the address and structure length of any instance of a structure (in general) must be a multiple of $K_{\max}$, i.e., $\alpha \cdot K_{\max}$.

Consider as an example, the following structure:

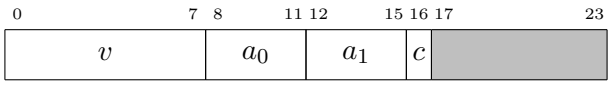
```
struct {
  double v;
  int a[2];
  char c;
} T2;
```

and its naïve alignment:



this works fine for internal purposes (note that each field of $T2$ is aligned along K_i), but is problematic if we wish to set up the following declaration: `T2 *arr`. Note that the $i > 1$-th element of this array will *not* be aligned.

As such, we introduce extra padding on the end of the structure in order to align it within self-referential arrays, along $K_{\max}$, which happens to be 8:



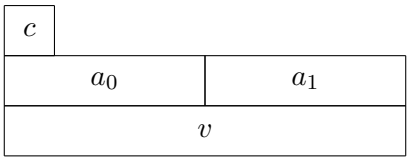
2.2 Unions

Structures provide a great way to access *multiple* members of a structure in a contiguous *block* of memory. Consider an alternate use-case: accessing the *same* block of memory under different type-signatures.

Unions provide a way to do just that. Consider the following example:

```
union {
  char c;
  int a[2];
  double v;
} U;
```

and it's layout in memory:



C (rightly) imposes the limit that you can access one of these at a time. All of the following declarations are therefore valid:

```
void example(U *u) {
  char c = u->c;
  int a[2] = u->a;
  double v = u->v;
  int x = a[0];

  // Note that all "members" of u share the same location.
  assert(((size_t *) c) == ((size_t *) a));
  return;
}
```

Chapter 3

Caches

Computers access memory frequently, and access at such high frequencies can often times be a hindrance to high performance. As such, we cache frequently accessed blocks of memory, taking advantage of two properties relating to locality:

Temporal locality Recently accessed memory is likely to be referenced again in the near future.

Spacial locality Items accessed in memory with low changes in address tend to be referenced close together in time.

Define the performance of a cache to be proportional to its Average Memory Access Time (AMAT), and defined as follows:

$$\text{AMAT} = \text{HT} + \text{MR} \cdot \text{MP}$$

Where each component is defined as follows:

HT The time to deliver a block from the cache to the processor.

MR How often memory references are not in the cached. Equal to total misses over total accesses, or $1 - \text{HR}$.

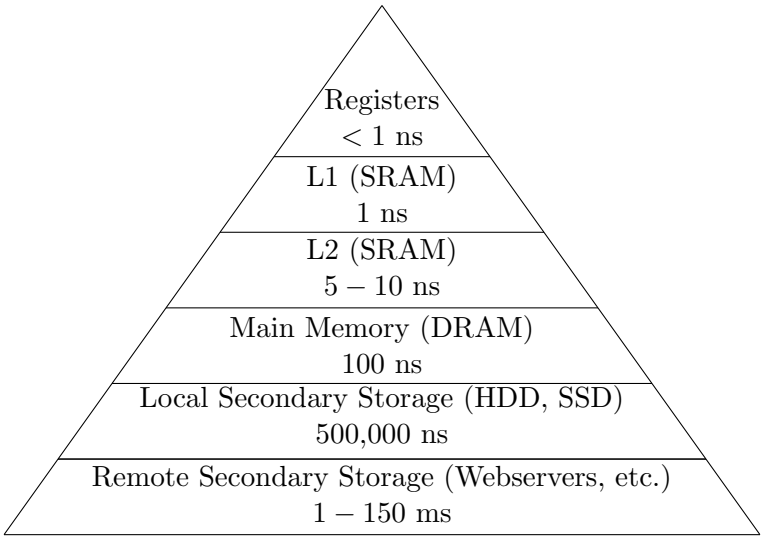
MP Additional time required to access and load out-of-cache memory.

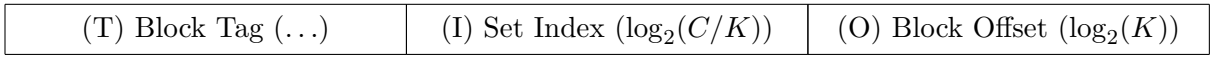
The principle of the above pyramid is as follows: further up the pyramid you go, the faster the memory storage device becomes. The cost for this increased speed is monetary and physical: the cost-per-storage-unit drives up while the total amount of storage per device goes down.

In other terms, you cannot fit a 64-bit word size address space in CPU registers, but you can fit a very small amount of it in a very small amount of space and access it very quickly. Further down the hierarchy allows you to store more of this information, but at slower access speeds, and cheaper cost.

3.1 Cache Organization

Block Size (K) A block is the unit of transfer between the cache and main memory. It is always given as a power of 2, and each block consists of adjacent bytes in memory. Within a block of memory, the specific byte is given by the block offset, which is k bits, or $\log_2(K) = k$.





Cache Size (C) The total cache size (C) is the total amount of data that the cache can store. It is either given as a storage unit, or in number of blocks (in which case, C/K). Note that the cache index (i.e., the set that a block belongs within is given as the next $\log_2(C/K)$ bits.

3.1.1 TIO-breakdown

The total address of a location in memory is mapped into the cache as follows. The block tag disambiguates between multiple blocks in the same set, and is denoted (T). The set index disambiguates which set within the cache the block belongs, and is denoted (I). The block offset (denoted (k)) lets a caller select which byte they want within a block.

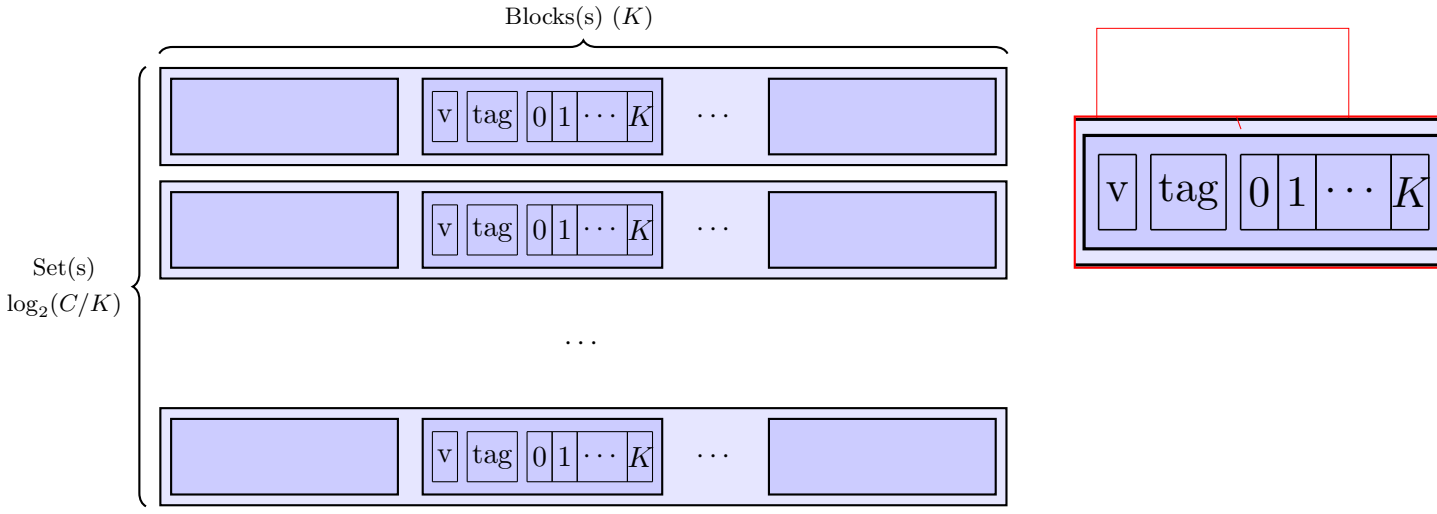


Figure 3.1: Typical cache layout

3.1.2 Associativity

We are met with a problem: what if two different locations map into the same block of the cache? If so, and we are alternating accesses between them, we will quickly run down into a 0% hit rate, negatively impacting performance.

So, we introduce the concept of associativity, the notion that multiple blocks can belong to the same set. For example, a "direct-mapped cache" is 1-way set associative, and could contain 8 blocks (meaning it would therefore have 8 sets, as well). A cache could also be 4-way set associative, meaning that for there to be 8 blocks, the cache would comprise of 2 sets each containing 4 blocks. Finally, a cache could be "fully associative", indicating that there would be 1 set with 8 blocks in it.

Associativity (E) Denote an E-way set associative cache as follows: we index into cache sets, of which there are $C/K/E$, and use the lowest $\log_2(C/K/E) = s$ bits of the block address.

Direct-mapped A direct-mapped cache will have $E = 1$, so $s = \log_2(C/K)$ (as before).

Fully assosciative A fully associative cache will have $E = C/K$, so $s = 0$ (i.e., there is only one set to choose from).

3.1.3 Eviction Policies

Now we arrive at the question: if there are no empty blocks, where should we cache a new block? To answer this, we must consider the value of E, the set-associativity of our cache. When E denotes a direct-mapped cache, there is no choice: the whole set must be evicted. But when $E \neq C/K$, we can evict *any* block within the set. Therefore, we could use least-recently used (LRU) or, more practically, "not most recently used".

3.1.4 Types of Cache Misses

There are three types of cache misses. They are as follows:

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

Conflict miss The cache was large enough, but multiple memory locations map to the same set/block.

Capacity miss Occurs when the set of active cache blocks is larger than the cache (i.e., wouldn't fit even if the cache was fully-associative).

3.1.5 Cache Writes

Given that there are now multiple copies of data, how do we ensure that write-able data is kept in-sync across the cache hierarchy? Recall that only the top-most layer of the cache is read from and written to, so we need not write back early.

Write-hit(s)

We essentially have two options on write-hits:

Write-through Immediately write to the next level.

Write-back Defer writing to the next level until the line is evicted.

Write-miss(es)

We have two options on write-misses, as well:

Write-allocate Load into cache, update the line in cache.

No-write-allocate Write immediately to memory.

Chapter 4

Control Flow, Processes

4.1 Motivation

An exception can be raised to transfer control from a running process to the operating system's kernel in response to some event event. Between a current and next instruction, an event can occur triggering an exception, which transfers control control flow to the operating system's kernel. The kernel can then process the event, and then either (1) return to the current instruction, (2) return to the next instruction, or (3) abort.

4.2 Processes

There are three types of exceptions; they are outline as follows:

Interrupts These are asynchronous interrupts, triggered by `SIGKILL`, or `SIGTERM`, i.e., pressing Ctrl-D, Ctrl-C on the keyboard.

Traps & Faults These are synchronous interrupts, like a page fault or divide by zero exception, etc.

4.2.1 Running Exceptions

4.2.2 Exception Table

How do we know which code to run in the kernel for a given exception? Note that there is a jump table for exceptions (known as the "Interrupt Vector Table", or IVT). Each entry in the table's index is given as the exception number k.

4.3 Asynchronous Exceptions (Interrupts)

These are caused by events external to the processor, like I/O interrupts (pressing keys on the keyboard, clicking the mouse, a packet arriving from the network, etc.) These can also be caused by timer interrupts.

4.4 Synchronous Exceptions

There are several kinds of exceptions that are caused by events that occur as a result of executing an x86-64 instruction:

Traps

These are intentional transfers of control from user-land to the kernel in order to perform a system call. Also covered are: breakpoint traps, special instructions, etc. Control is returned from a trap to user-space's next instruction.

Faults

These are unintentional faults and transfer control to the kernel, but may be recoverable. This occurs when a page fault, divide-by-zero, or segmentation faults occur. The handler either re-executes the current instruction, or aborts.

Aborts

These are unintentional and unrecoverable.

4.5 Processes

A process is another abstraction provided by the operating system to maintain the interface between the program and the underling hardware. This allows the operating system to provide the illusion that–when seen from the program's point of view–allows the program to "believe" that it is the only program running on that machine.

This is the result of an inherent limitation of our CPU(s), the number of logical cores on the silicon. A CPU can only run so many programs concurrently, and thus it is left up to the operating system to "juggle" the running process list and make it appear as if all programs are running concurrently *even though they are certainly not.*

A process provides each program with two key abstractions: (1) logical control flow and (2) private address space.

4.5.1 Context Switch(-ing)

How do we transition from one running process to the next? Inherently, we must capture the state of a running process that is about to be idled, and save that state somewhere. Then, before resuming the next process, (i.e., the process that we are switching into) we must load the frozen state of that process, and restores that, then resume running the program.

At a finer level of detail, this appears as the following:

1. Save current registers in memory.
2. Schedule the next process for execution.
3. Load the saved registers and switch address space.

Recall that the context switch is performed by the kernel, so we must first interrupt into the ring-0, and then the kernel can perform the context switch.

4.6 Creating Processes

When a program wants to launch another program, they execute the following two system calls in order:

`fork()` Create a copy of the current process running in a new PID-space.

`exec*()` Replace the calling process with another process's code and address space. (Including: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`).

There are an additional list of process-related system calls, which is as follows:

`getpid()` Get the PID of the calling process.

`exit()` Tell the current process to exit itself.

`wait()`, `waitpid()` Wait for the current or another process to exit.

4.6.1 `fork()`

`fork()` is a system call with the following signature:

```
pid_t fork(void) { /* ... */ }
```

Calling `fork()` creates a new child process that is identical to the calling process, with one exception. The `pid_t` returned to the child process is equal to 0, whereas the return from the parent process is that of the child process.

In the child process, a new virtual address space has been created. Note that this function can be confusing, since it is called once but returns twice, unlike... most other functions.

4.6.2 `exec*()`

`exec*()` is a system call with the following signature:

```
int exec*(char *path, char *argv[], char *envp[]) { /* ... */ }
```

it replaces the current process's code and address with the code of a different program.

Chapter 5

Virtual Memory

5.1 Motivation

Given a word size w, we must allocate $\ll 2^w$ bytes of memory to project that each running process has all of memory space in such a way that each process can not see other process(es) running memory.

To remedy this problem, we introduce *virtual memory* (VM) as a way of dividing physical memory into virtual locations useable by running processes in such a way that satisfies these constraints.

Pages Note that a page is a contiguous section of memory that is loaded and unloaded into virtual memory as an atomic unit. The pages are loaded and stored from physical memory, and are occasionally "swapped" onto the hard-disk (or in descending order down the caching hierarchy.)

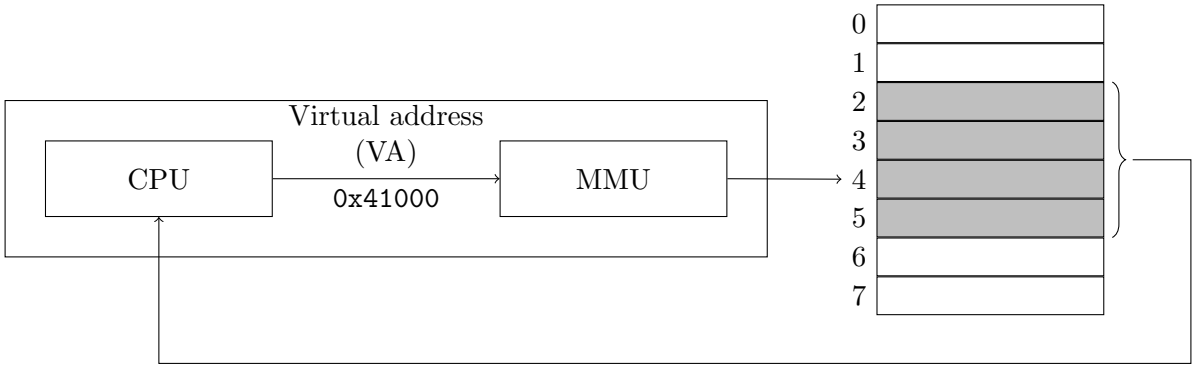


Figure 5.1: CPU Chip containing a MMU

5.2 Address Translation

So, how do we perform the role of the MMU in the figure above? Or, how do we translate from an in-process virtual address to an out-of-process physical address?

1. Use the page table to translate a virtual page number into a physical page number.

2. Re-use the virtual page offset for the physical page offset.

3. Read that location from memory.

5.2.1 Physical Page Lookup

We split the virtual address into the following:

Virtual Page Number (VPN)	Page Offset

First, we try and find the virtual page number in the Translation Look-aside Buffer (TLB), which maintains a cache from virtual page numbers to physical page numbers. If the TLB contains a cache entry for the virtual page number we were interested in, stop here. If not, we seek ahead from the page table base register (PTBR) to the correct page number, and look for a physical page number associated with it. If the page table didn't contain this entry, we then fault.

5.2.2 Page Faults

If we encountered a page fault (i.e., because we hadn't loaded a portion of memory into the cache, or because we are swapping), we raise a page fault exception, which transfers control into the kernel, which will then create a page and load it into memory, returning to the instruction that raised that fault.

The kernel must select an eviction target (if we were swapping) which it then pages out to lower in the cache hierarchy, and then pages in the correct bytes.

5.2.3 Overview

1. Check Translation look-aside buffer (VPN $\mapsto$ PPN)

   TLB hit Fetch translation, return the PPN

   TLB miss Check page-table in memory

      Page-table hit Load the entry from the page table into the TLB

      Page fault Fetch the page from disk to memory, updating the page-table entry in cache, and the loading the entry into the TLB

2. Check cache (PA $\mapsto$???)

   Cache hit Return data to processor

   Cache miss Fetch data from memory, cache it, return to processor.

Recall the following terminology:

5.2.4 Basic Parameters

$N = 2^n$ Number of addresses in virtual address space.

$M = 2^m$ Number of address in physical address space.

$P = 2^p$ Page size (in bytes).

5.2.5 Components of virtual address (VA)

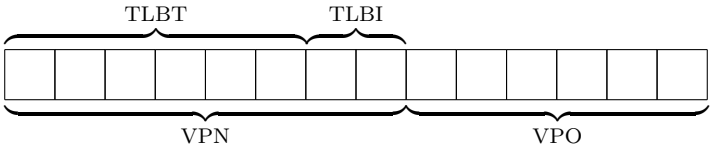


Figure 5.2: Layout of 14-bit virtual memory.

VPO Virtual page offset. VPO = p (i.e., number of bits needed to represent all bytes in a page).

VPN Virtual page number. VPN = $n-p$ (i.e., the remaining bits in a virtual address not consumed by the VPO).

TLBI TLB index. TLBI = $\log_2(E_t)$ (i.e., number of bits needed to represent all blocks in a set in the TLB).

TLBT TLB tag. (rest)

5.2.6 Components of physical address (PA)

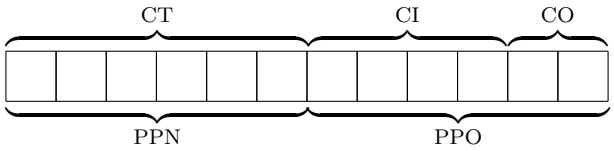


Figure 5.3: Layout of 12-bit physical memory.

PPO Physical page offset (identical to VPO)

PPN Physical page number

Chapter 6

Memory Allocation

Recall that there are multiple ways to store program data. We can store (1) static global data, which is of fixed size and has a lifetime occupying the entirety of the running program. We can store (2) stack-allocated data, which are used to store local or temporary variables, and are dead on return. We can store (3) heap allocated data, the size of which is known only at runtime, and the lifetime is known also only at runtime.

So, we need a way to dynamically create and destroy blocks of memory on the heap. This is known as Dynamic Memory Allocation (DMA), and there are two sub-types which we consider here:

Explicit allocator The programmer must allocate and free space themselves.

Implicit allocator The programmer only allocates space, and garbage collection frees space automatically.

Within the heap, a DMA will allocate memory as a collection of variable-sized blocks, each of which is either allocated or free.

6.1 Memory Allocation in C

To allocate and free memory in C, one must `#include <stdlib.h>` in order to introduce the signatures `malloc()` and `free()`, which we describe below.

6.1.1 `malloc()`

```
void *malloc(size_t size) { /* ... */ }
```

`malloc()` allocates a continuous block of `size` uninitialized memory. It returns a pointer to the beginning of the block, or `NULL` if the memory allocation failed. Typically these are aligned to 8- or 16-byte boundaries, and are not necessarily adjacent.

Related, there exist two other functions:

```
void *calloc(size_t nr, size_t size)  { /* ... */ }
void *realloc(void *ptr, size_t size) { /* ... */ }
```

Where `calloc()` "zeros out" memory, and `realloc()` resizes a previously initialized block.

6.1.2 `free()`

```
void free(void *ptr) { /* ... */ }
```

`free()` frees the entire block of memory pointed to by `p`, which must be a value originally returned by either `malloc()`, `calloc()`, or `realloc()`.

6.2 Performance Goals

Given a series of memory requests (i.e., calls to either `malloc()` or `free()`) denoted $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$. We wish to maximize both throughput and peak memory utilization.

6.2.1 Throughput

Let "throughput" be defined as the numer of completed requests per unit time.

6.2.2 Peak Memory Utilization

Let P_k be the aggregate payload of the k-th request, which is given to be the sum of all currently allocated payloads.

Let H_k be the size of the heap at the time request k is issued.

$$U_k = \frac{1}{H_k} \cdot \left(\max_{i \leq k} P_i \right)$$

6.3 Fragmentation

One way that we can not maximize U_k is by introducing fragmentation. Recall that fragmentation is occupied space in the heap H that is not used by any calling code. This can occur because of structure fragmentation, which is space necessary for structure alignment, but can also occur between members of structures.

We now continue our discussion of fragmentation considering only fragmentation as it pertains to the heap.

6.3.1 Internal Fragmentation

For a given block, let the internal fragmentation of that block be the space (if any) left over by unused space within that block. This can be because of padding, overhead of maintaining heap structures, or because of explicit policy decisions.

6.3.2 External Fragmentation

External fragmentation is when the series of requests R leaves "holes" between blocks.

6.4 Implicit free list

An implicit free list uses the length of each blocks to link all blocks in a doubly-linked list.

Let each block be defined as follows:

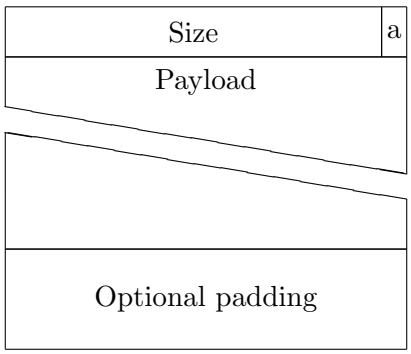


Figure 6.1: Implicit free list general block format

Because each block is aligned, we know that a quantity of the low order bits will be always 0. Therefore, we can use them as we wish, and in this case we use the lowest bit of the size field to store a, which is true iff the block is allocated.

We now describe several algorithms for finding free blocks in an implicit free list:

6.4.1 First fit

This algorithm searches from the beginning and chooses the first free block that fits. It is implemented as follows:

```
void *first_fit(void *heap_start) {
  void *p = heap_start;
  // While we are (1) in range (2) allocated, and (3) too small:
  while ((p < end) && ((*p & 0x1) || (*p <= len))) {
    p += (p* & ~0x1);
  }
  return p;
}
```

6.4.2 Next fit

This algorithm works in the same way that first fit does, but takes as the starting index the "last-allocated" block instead of the total `heap_start`, but in practice results in worse fragmentation.

6.4.3 Best fit

This algorithm chooses the best free block, large enough to contain the block and retaining the fewest left-over bytes (ideally 0). This keeps the fragmentation low, but results in often worse throughput.

6.4.4 Allocating a Free Block

When we allocate a free block, we must first split that block such we have *two* free blocks, one about to be allocated, and the other the remaining bytes not allocated by the current call to `malloc()`.

The implementation is as such:

```
void split(void *ptr, size_t n) {
  // Round to the nearest eight (2^3).
  int new_size = ((n+7) >> 3) << 3;
  // Note the current size of the block to-be-shrunk.
  int old_size = *ptr;
  // Shrink the block.
  *ptr = new_size;
  if (new_size < old_size) {
    // If there is enough space to allocate a new block, do so:
    *(b+new_size) = old_size - new_size;
  }
}
```

6.4.5 Freeing an Allocated Bloc

The simple implementation is as follows:

```
void free(void *ptr) { *(p-4) &= ~0x1; }
```

But, this leads to "false fragmentation", a situation that occurs when two free blocks are present adjacent to each other. This can have the effect of "tricking" the memory allocator into thinking that there isn't enough space to allocate a block since the first free block doesn't appear large enough, when the two put together in fact are sufficient.

We therefore introduce the concept of coalescing with the next block if it is also free. However, we are met with the problem of bidirectional coalescing! What if we free the block *after* a free block, instead of before?

To remedy this problem, Knuth introduces the concept of boundary tags, which replicate the header at the bottom of free blocks, which means that we can traverse backwards one block and examine the preceding block:

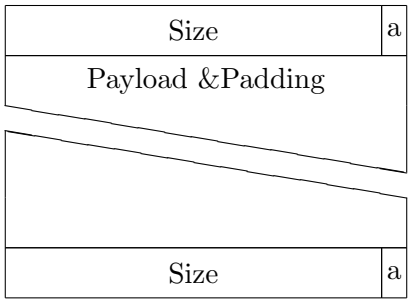
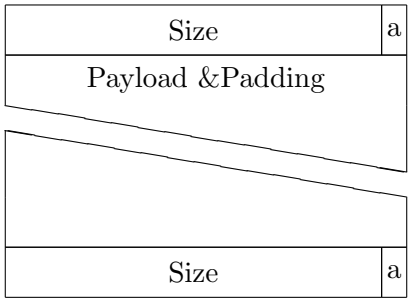


Figure 6.2: Implicit free list bi-directional block format

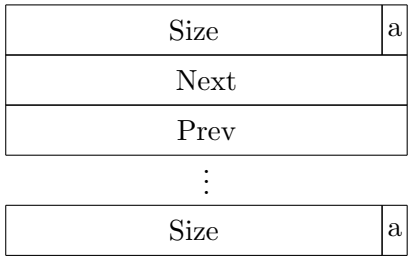
6.5 Explicit free list

The implicit free list is a satisfactory model of memory allocation, but is problematic because in order to find a free block in the list, we must traverse over all of the blocks (including those that are allocated). In order to fix this, we create an explicit linked list over all free blocks, which is more complicated to maintain, but only by a margin, and results in worthwhile performance improvements.

We describe the block layout now:



(a) Explicit free list allocated block format



(b) Explicit free list free block format

Figure 6.3: Free and allocated blocks general format

6.5.1 Linkages

We have a few choices for our "next" and "prev" pointers. Since they are just that (`void *`, in practice), they can point *anywhere* in the heap, not just to the physically next or last block.

Linear doubly-linked list In a linear doubly-linked list, we must take in a head pointer, and let the first node's previous pointer be `NULL`, as well as the last node's next pointer. This is a great choice for first- and best-fit.

Circular doubly-linked list In a circular doubly-linked list, we must have pointers to determine which node we are starting with, but there are no `NULL` pointers, since the terminating condition is to go back to the starting point. This is a great choice for next- and best-fit.

In either of the above two (linear and circular doubly-linked lists), we have the choice whether to link blocks next to each other in the heap, or whether we can retain any ordering on the blocks. The prior is known as a "logically" linked list, whereas the later is known "physically" linked list.

6.6 Segregated free list

We describe this and the next section in much shorter detail. In a segregated free list, we denote each size class of blocks and give it its own free list, such that the heap is organized to be an array of free lists.

This allows each sub-list to retain far fewer amounts of fragmentation, and allocate correctly-sized blocks much quicker. In order to allocate a block of size n, we apply the following:

1. Search for an appropriate free list block of size $\exists m.m \geq n$.

2. If a such m is found, we split the block and place the remaining fragment on the appropriate list.

3. If no block is found, we increase $m \rightarrow m'$.

4. If no block was found in the first place, we issue a system call `sbrk()` to grow the size of the heap, and then try again.

In order to free a block, we apply the following:

1. Mark the block as free

2. Coalesce as needed (or avoid doing so)

3. Place the remaining block(s) on appropriate size lists.

6.7 Garbage Collection

Garbage collection is a term used to describe the automatic reclamation of heap-allocated storage, such that an application programmed atop a runtime implementing automatic memory management never has to explicitly free its own memory.

The problem is that we do not know in general when memory is truly able to be freed, but we can tell that certain blocks are unreachable and therefore are likely free candidates.

However, the memory allocator does not know which values are and are not pointers into other locations in the heap, so we must get assistance from the compiler.

6.7.1 Memory as a Graph

Let each heap block be a node (such that the total set is V) in a graph. Let each pointer along that graph (such that $P \subseteq V \times V$). Locations not in the heap that contain pointers into the heap are called root nodes, and can comprise of registers, stack locations, global variables, and etc.)

We denote a block as "reachable" if there exists a path from any root node to that node. We denote all other blocks as "garbage" when they are non-reachable.

6.7.2 Overview

We describe the Mark-and-sweep algorithm (McCarthy, 60) which uses an extra low-order bit as the mark bit. We then "mark" all root nodes, traversing the edges outgoing P_{n_i}, and then "sweep" over all remaining non-marked nodes.

6.7.3 `mark()`

Let the procedure `mark()` be implemented as follows:

```
void mark(void *ptr) {
  int i;

  // Do nothing if we aren't given a pointer, or if our pointer is
  // already marked.
  if (!(is_pointer(ptr)) { return; }
  if (*ptr & 0x2) { return; }

  // Set the mark bit.
  *ptr |= 0x2;

  // Traverse through all pointers (aligned along word boundaries) in the
  // current block marking all of them as reachable.
  for (i = 0; i < len(p); i++) {
    // Interpret the next size_t bytes in memory as a pointer.
    mark((void *) ((size_t *) (((char *) p) + sizeof(void *))));
  }
}
```

6.7.4 sweep()

Let the procedure `sweep()` be implemented as follows:

```
void sweep(void *ptr, void *end) {
  size_t plen;
  while (ptr < end) {
    plen = len(ptr);
    if (*ptr & 0x2) {
      *ptr &= ~0x2;
    } else if (*ptr & 0x1) {
      free(ptr);
    }
    p += plen;
  }
}
```

Chapter 7

Java

So far, we have seen how programming and systems-level concepts work in C. We have covered representation of data, pointers, references, casting, function calls, runtime environment, and the translation from high- to low-level code. Let's observe now what those concepts look and feel like when translated to Java.

7.1 Data in Java

7.1.1 Primitives

Many of the primitive data types that we saw in C translate rather directly to Java. For instance, "pointers" in C are called "references" in Java, and pointer arithmetic is no longer allowed, but the conception is mostly the same. Portability is guaranteed in Java in a way that isn't possible in C, for instance Java's `int` type is the same size regardless of machine.

7.1.2 Arrays

Every element of an array is initialized to the "zero-value", and the length is specified by an immutable field at the start of the array. This allows us to do the following:

```
private static void example() {
  int[] arr = new int[5];
  assert(5 == arr.length);
}
```

Note that this allows us to determine that array index operators (lookups, assignments) are bounds checked. C does *not* provide this guarantee.

7.1.3 Strings

Java has three distinctions when it comes to storing a string of characters.

1. Characters are stored as 2-byte UTF-8 codepoints instead of ASCII.
2. Strings are *not* bounded by a NUL character (they instead include a length header).
3. Strings are read-only.

7.1.4 Objects

Sub-members of objects in Java are always stored as references, and never stored inline. (Let "complex" data-types that are affected by this change include arrays and other objects).

Member accesses Recall in C we had the `->` operator (which made `r->m` equivalent to `(*r).m`). In Java, *all* objects are references, so the `.` operator in C *does not exist* in Java, hence *all* member accesses are in-effect `->`'s.

Object References Recall also that in C, pointers can point to any location and that references are an analogue of pointers in Java. References, notably, cannot store the location of an arbitrary location in memory, and instead can only store the location pertaining to the start of an object.

Casting In C, we are allowed to "cast" from one type into another arbitrarily. This allows data that was stored as one type to be interpreted as another type, regardless of the declared type of that location in memory. In Java, we are not permitted to cast over such a wide domain, instead, we are restricted to casting *above* our class C in the class hierarchy.

7.2 Methods & Method Dispatch

Note that each instance of a Java object that contains a header, a pointer to the `vtable`, and then includes the contents of the fields.

Header What information is contained in the header? The header contains information useful for garbage collection, hashing, locking, and etc.

vtable What is the vtable? The vtable is in essence a jump table for virtual methods and other class information. There exists one vtable per class.

7.2.1 Constructors

When we call a constructor in Java (invoked via the `new` keyword), the Java runtime allocates enough space for the object, initializes all fields to their zero-value (chosen according to the kind of their type), and then runs constructor methods.

7.2.2 Methods

Recall that a static method takes no receiver, and are like functions. Recall further that instance methods take a receiver (called `this`), and can be overridden in subclasses. When a method is called, the code that actually executes the code contained within that method is chosen at runtime by looking up the pointer in the vtable.

Concretely, if we write the following in Java:

```
p.samePlace(q);
```

it would be translated as follows into C:

```
p->vtable[1](p, q);
```

7.2.3 Subclassing

Amazingly, when we subclass a class from an existing one (such that $\tau' \prec \tau$) the fields in τ' are located *after the fields in* τ. This means that, in practice, we can use methods contained in the vtable for τ on an instance of a τ'.