CS472 Assignment 3

Mark Bereza

November 17, 2017

Part 1

CS472 Assignment 3 — MARK BEREZA

Part 1

1. 4kb page → 4 · 1024 bytes = $2 \cdot 2^{10}$ bytes = 2^{12} bytes
   32-bit address space → 2^{32} unique addressable bytes
   Total number of pages = $2^{32}/2^{12} = 2^{20}$
   Required memory = $2^{20} \cdot 10$ bytes = 10485760 bytes
   = 10240 KB = **10 MB**

2. 64-bit address space → 2^{64} unique addressable bytes
   Total number of pages = $2^{64}/2^{12} = 2^{52}$
   Required memory = $2^{52} \cdot 10$ bytes = 45035996273704960 B
   = 43980465111040 KB = 42949672960 MB = 41943040 GB
   = 40960 TB = **40 PB**

3. Twice the page size means half as many entries to address the same space and thus half the amount of memory to store the page tables. So 10 MB/2 = **5 MB**

4. Using the same reasoning as part 3, but applied to the result from part 2: 40 PB/2 = **20 PB**

5. Pipelining is useful because it allows a processor to start the new instructions while the current instruction is still being processed for instruction that take several clock cycles. This drastically increases throughout, allowing instructions that would normally take many cycle to be completed one per cycle assuming the pipeline remains full.

6. The IA32e paging structure is a 4-level paging structure utilized on some Intel architectures and is one approach to 64-bit paging.
IA32e paging is enabled on an Intel CPU only if the following four conditions are met:
1. The processor supports the Intel 64 architecture
2. Paging is enabled (bit 31 of CR0 is set)
3. Physical address extension (PAE) is enabled (bit 5 of CR4)
4. 4 level paging mode is enabled (bit 8 of the IA32_EEER_MSR is set)

This paging structure maps 48-bit virtual address to 52-bit physical addresses, meaning the maximum addressable space is 2^{48} bytes = 256 TB. As for the page dimensions, each page in IA32e is 4KB, each, each table entry is 8 bytes, and each table has 512 entries. The first paging structure's location is stored in CR3, but every other structure's address is calculated. In addition to containing pointers to either child paging structures or a physical memory page, entries also contain information about memory access permissions, whether the page has been used for translation, and other metadata. As for the page levels themselves, they are: PML-4 table, Page Directory Pointer Table (PDPT), Page Directory (PD) and Page Table (PT). Each PML-4 entry can address 512 GB, each PDPT entry can address 1 GB, each PD entry can address 2MB, and, obviously, each PT entry can address 4KB. To reference each of these layers the virtual memory addresses are segmented. The fist 16 bits are for mode mapping, the next 9 are for PML-4, the next 9 are for PDPT, the next 9 are for PD, the the next 9 are for PT, and the final 12 are for the page offset.

Part 2

"Memory Optimization" Summary

This presentation makes an argument for why memory optimization is something programmers should be aware of and try to attain in their code. The author claims that the reasons why it is important for programmers to write cache-aware code are that memory access speeds have increased slowly compared to CPU clock speeds, the cache is generally under utilized, and compilers are generally poor at figuring out how to optimize memory accessing on their own.

The author then describes the various reasons why caches might "miss" and goes into detail describing strategies to improve one's code to reduce these misses and increases performance overall by making better use of the cache. These strategies include profiling one's code to get a better picture of when and where memory accesses are occuring, reordering code to improve locality, and compressing/padding code to improve use of cache lines. In particular, the presentation touches upon prefetching/preloading data that will be used in an anticipatory fashion, reordering fields in structures/objects based on actual use and hot vs cold members, and cache-aware approaches to tree-like data structures. The author further suggests the use of memory pools and linearization of hierchial data to increase spatial locality and thus incrases cache hit rate. Finally, the author discusses the cache performance penalties associated with aliasing (multiple indirect references to the same memory location) and higher levels of abstraction, which both make it harder for the compiler to optimize read accesses. The rest of the presentation is devoted to detailing several strategies for minimizing aliasing, including consuming pointer variables into distinct local variables, avoiding class abstraction where possible, making variables of incompatible types explicitly so to the compiler, and leveraging C's 'restrict' keyword.

"What Every Programmer Should Know About Memory" Summary

Our interest in this paper has to do with computation of cache sizes. First, in section 3.3.1, the paper states that the total cache size may computed using the following formula:

$$\text{cache size} = \text{cache line size} \cdot \text{associativity} \cdot \text{number of sets}$$

Beyond that, section 6.2.3 describes how to compute a safe lower limit for cache size on a Linux machine. This involves reading
`/sys/devices/system/cpu/cpu*/cache/size` and dividing "the numeric value contained within by the number of bits set in the file `shared_cpu_map`."

Alternatively, an easier method presented in the paper involves calling `sysconf()` with a macro defining the cache to be queried passed as its only argument.

Part 3

For this part, I simply inspected the contents of the files located under /sys/devices/system/cpu/cpu0/cache/ to determine the number of caches, their level, their type, their line size, their number of sets, and their ways of association. Here, I made the assumption that these properties would be the same across the different cores and thus querying a single core's data would be sufficient.

Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <stdint.h>
#include <sys/types.h>
#include <string.h>

#define CACHE_PATH "/sys/devices/system/cpu/cpu0/cache/"

/* Referenced https://stackoverflow.com/questions/1121383/counting-the-number-of-files-in-a
uint8_t cache_count() {
    uint8_t count = 0;
    DIR* cache_dir;
    struct dirent* entry;

    cache_dir = opendir(CACHE_PATH);
    if (cache_dir == NULL) {
        fprintf(stderr, "Error: could not open cache directory.\n");
        exit(-1);
    }
    while ((entry = readdir(cache_dir)) != NULL) {
        if (strcmp(entry->d_name, ".") && strcmp(entry->d_name, "..")) {
            count++;
        }
    }
    closedir(cache_dir);

    return count;
}

char* cache_type(uint8_t i) {
    static char type[64];
    char filename[256];
    FILE* f;

    sprintf(filename, "%s%s%u%s", CACHE_PATH, "index", i, "/type");
    f = fopen(filename, "r");
    if (f == NULL) {
        fprintf(stderr, "Error: could not open cache type file.\n");
        exit(-1);
    }
    memset(type, 0, sizeof(type));
    fscanf(f, "%s", type);
    fclose(f);
```

```
    return type;
}

uint8_t cache_level(uint8_t i) {
    uint8_t level;
    char filename[256];
    FILE* f;

    sprintf(filename, "%s%s%u%s", CACHE_PATH, "index", i, "/level");
    f = fopen(filename, "r");
    if (f == NULL) {
        fprintf(stderr, "Error: could not open cache level file.\n");
        exit(-1);
    }
    fscanf(f, "%hhu", &level);
    fclose(f);

    return level;
}

uint64_t cache_line_size(uint8_t i) {
    uint64_t line_size;
    char filename[256];
    FILE* f;

    sprintf(filename, "%s%s%u%s", CACHE_PATH, "index", i,
            "/coherency_line_size");
    f = fopen(filename, "r");
    if (f == NULL) {
        fprintf(stderr, "Error: could not open cache line size file.\n");
        exit(-1);
    }
    fscanf(f, "%lu", &line_size);
    fclose(f);

    return line_size;
}

uint64_t cache_sets(uint8_t i) {
    uint64_t sets;
    char filename[256];
    FILE* f;

    sprintf(filename, "%s%s%u%s", CACHE_PATH, "index", i,
        "/number_of_sets");
    f = fopen(filename, "r");
    if (f == NULL) {
```

```
        fprintf(stderr, "Error: could not open cache sets file.\n");
        exit(-1);
    }
    fscanf(f, "%lu", &sets);
    fclose(f);

    return sets;
}

uint64_t cache_ways(uint8_t i) {
    uint64_t ways;
    char filename[256];
    FILE* f;

    sprintf(filename, "%s%s%u%s", CACHE_PATH, "index", i,
        "/ways_of_associativity");
    f = fopen(filename, "r");
    if (f == NULL) {
        fprintf(stderr, "Error: could not open cache ways file.\n");
        exit(-1);
    }
    fscanf(f, "%lu", &ways);
    fclose(f);

    return ways;
}

uint64_t cache_size(uint8_t i) {
    return cache_line_size(i) * cache_ways(i) * cache_sets(i) / 1024;
}

int main() {
    uint8_t num_caches = cache_count();
    printf("Number of caches:              %u\n\n", num_caches);
    for (int i = 0; i < num_caches; ++i) {
        printf("Cache %d level:                %u\n", i, cache_level(i));
        printf("Cache %d type:                 %s\n", i, cache_type(i));
        printf("Cache %d line size:            %lu\n", i, cache_line_size(i));
        printf("Cache %d ways of associativity: %lu\n", i, cache_ways(i));
        printf("Cache %d number of sets:       %lu\n", i, cache_sets(i));
        printf("Cache %d size:                 %luK\n\n", i, cache_size(i));
    }

    return 0;
}
```

Part 4

For this part, I simply wrote a function that would reverse the order of bytes for any byte array. Then, I used a preprocessor directive to determine the endianness of the system. If the system was determined to be little-endian, byte swapping was performed. Otherwise, the data was left as-is.

Source Code

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

void byte_swap(char* bytes, int len) {
    int num_swaps = ceil((len - 1) / 2.0);
    for (int i = 0; i < num_swaps; ++i) {
        int back = i;
        int front = len - 1 - i;
        char temp;
        temp = bytes[back];
        bytes[back] = bytes[front];
        bytes[front] = temp;
    }
}

int main(int argc, char **argv)
{
    short val;
    char *p_val;
    p_val = (char *)&val;
    p_val[0] = 0x12;
    p_val[1] = 0x34;

#if __BYTE_ORDER__ == __ORDER_BIG_ENDIAN__
    printf("System uses big-endian memory architecture.\n");
#elif __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    printf("System uses little-endian memory architecture.\n");
    byte_swap(p_val, 2);
#else
#error "Endianess cannot be determined at compile time."
#endif
    printf("%x\n", val);

    return 0;
}
```

Part 5

For this part, I change the `byte_swap()` function to first determine the endianness of the system at runtime by storing `0x0102` to the integer portion of a union and checking whether the first byte (via array indexing) was 1 or 2. If the first byte was 1, then the system was big-endian. Otherwise, it was little-endian. As with Part 4, byte swapping was performed if the system was determined to be little-endian.

Source Code

```
#include <stdio.h>
#include <math.h>
#include <stdint.h>

/* Referenced https://stackoverflow.com/questions/1001307/detecting-endianness-programmatic
void bi_endian(char* bytes, int len) {
    union {
        uint16_t u;
        char c[2];
    } test_val = {0x0102};

    if (test_val.c[0] != 1) {
        printf("System uses little-endian memory architecture.\n");
        int num_swaps = ceil((len - 1) / 2.0);
        for (int i = 0; i < num_swaps; ++i) {
            int back = i;
            int front = len - 1 - i;
            char temp;
            temp = bytes[back];
            bytes[back] = bytes[front];
            bytes[front] = temp;
        }
    } else {
        printf("System uses big-endian memory architecture.\n");
    }
}

int main(int argc, char **argv)
{
    short val;
    char *p_val;
    p_val = (char *)&val;
    p_val[0] = 0x12;
    p_val[1] = 0x34;
    bi_endian(p_val, 2);
    printf("%x\n", val);
```

```
    return 0;
}
```