ECE 472 Homework 3

Drake Vidkjer

November 21, 2017

1 Part 1: Create a document with answers for the following

For a 4KB page, and a 32 bit address space, calculate the amount of memory needed to store a process's page tables. Assume each entry in the page table requires 10 bytes. Show all calculations. 4KB page means 2^{12}, 32 bit address means 2^{32} so page table size = $(2^{32}/2^{12}) * 10$ = 10MB.

For a 4KB page, and a 64 bit address space, calculate the amount of memory needed to store a process's page tables. Assume each entry in the page table requires 10 bytes. Show all calculations. 4KB page means 2^{12}, 32 bit address means 2^{64} so page table size = $(2^{64}/2^{12}) * 10$ = 45035.99TB.

For a 8KB page, and a 32 bit address space, calculate the amount of memory needed to store a process's page tables. Assume each entry in the page table requires 10 bytes. Show all calculations. 4KB page means 2^{13}, 32 bit address means 2^{32} so page table size = $(2^{32}/2^{13}) * 10$ = 5.24MB.

For a 8KB page, and a 64 bit address space, calculate the amount of memory needed to store a process's page tables. Assume each entry in the page table requires 10 bytes. Show all calculations. 4KB page means 2^{13}, 32 bit address means 2^{64} so page table size = $(2^{64}/2^{13}) * 10$ = 22517.99TB.

Describe the concept of pipelining, and why it is useful. Pipelining is the process of somewhat parallelising the tasks performed by a CPU. Without pipelining, a CPU pulls an instruction from memory, then executes it all the way through, then pulls the next one from memory. Whith pipelining, the CPU pulls the instruction from memory, then as it starts executing the instruction, it pulls the next one from memory as well. and it continues to do this so that each component insited the CPU, is working on an instruction. This helps to allow the CPU to perform more tasks quicker, as it doesnt have to wait for a process to be completed before moving on to the next task.

Describe the IA-32e paging structure, in detail. IA-32e Paging works to translate linear address using paging structures. It works to translate 48-bit linear address to 52-bit physical address allowing for 256TB of linear address space to be used at one time. IA-32e paging uses a hierarchy of paging structures to translate address, and the first page structure is located using CR3. IA-32e can map linear addresses to 4KB, 2MB, or 1Gb pages depending on settings. The way IA-32e paging works is the same as other paging methods discussed in class. When attempting to translate a linear address used by a program to a physical address, the computer starts by looking at the first page in the series, for instance on linux called the page global directory, it then finds the correct entry which is representative of another page table, which the computer then goes to and looks in there for the next needed entry, it then does this all a third time in the page table entry and finds the correct entry. This is then representative of the physical address that is needing to be translated.

2 Part 2: Please write a short summary of each paper.

Memory Optimization The powerpoint, Memory Optimization, discusses how to optimize program by using processor cache. It talks about the pros and cons of using cache, including the three Cs and three Rs. The powerpoint goes on to talk about details of how to increase cache optimization by using decreased size to increase simplicity, and keeping in mind locality when linking during the compiling process. It also encourages monolithic functions and discourages OOP as it is less cache friendly. The presentation then goes on

ti give code examples of preloading/prefetching as well as analysis of caching performance and tree structures. Throughout the whole presentation, it talks about aliasing and how to avoid it. This includes multiple references to the same storage location, or missing optimization opportunities.

What Every Programmer Should Know About Memory The second of the two articles discusses the basics of pages and how they are implemented. It first talks about a simple paging example with a single page file that translates virtual to physical address. It talks about how this is not a practical implementations, as each page file has to generally be 4MB in size for each process, tying up a large amount of main memory just for page tables. The document then goes on to discuss a better way of doing pages, that is to have a page hierarchy. It discusses how this allows the different levels to be sparse and compact leading to less memory usage. On most systems the page hierarchy is either 4 or 3 levels, and on most x86 and x86-64 machines the process that utilizes these hierarchies, called page tree walking is done at the hardware level.

3 Part 3: Implement some C code to calculate the cache sizes of any arbitrary processor.

Reflection It took me a while to think about how to impliment this, and at first I was a little lost. But after reading the documents a little more, and doing some more research I ended up deciding to impliment this using access timing. Doing this tests the time it takes for a program to step through an array. The step size is fixed but the array size changes. One way to use this information to decide the cache size is to average the step time, and see when the average starts to change. I was originally going to measure the individual step time, but it was too quick, so I decided to instead average the step time and increment the array size to the signal to noise ratio will be lower. I was going to have it check for when the access times started to get larger, but I decided it would be better ot have the user do this, as it will be more accurate. So to find the cache size, you look at the step times, and when they start to get larger than the previous times, you know that the number of bytes that were stepped through the array is the size of the

cache.

4 Part 4: Modify the code so that it uses byte swapping and compiler directive to transform the code to be endian-neutral.

Reflection This implementation of endian neutralness is fairly easy, as it just relies on a compiler if/else statement to determine what endianess a system has and then compile only the relevant code for that system. In my opinion this is probably the cleanest way to implement endian neutralness, though it is a little more involved as youre basically writing two separate programs and then have to make sure compatibility between the two as well. But in this example, it is quick and easy.

5 Part 5: Modify the program from part 4 to use byte swapping and a runtime test (instead of compiler directive) to test for the endianness of the system that the code is running on.

Reflection I actually came up with this idea as the way to test endianness while implementing the first endian neutral code. I decided that I could test if it initializes the variable correctly assuming one endianness, and if that assumption is correct, then that means that the system follows that assumed endianness. It gives a little bit of overhead, especially with respect to this program because it is so large, but for a really large program, it would be a very small thing.

References

[1] `https://stackoverflow.com/questions/16323890/calculating-page-table-size`

[2] https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-a

[3] http://www.it.uu.se/edu/course/homepage/avdark/ht11/slides/11_Memory_and_optimi

[4] http://www.akkadia.org/drepper/cpumemory.pdf

[5] http://realtimecollisiondetection.net/pubs/GDC03_Ericson_Memory_Optimization.pp

[6] https://stackoverflow.com/questions/8978935/detecting-endianness