BYU Code

People may get confused on how to use the BYU code. File byu_tracereader.c can be compiled into your program like any other file in a multifile compilation. You will need to call NextAddress to get the next address. Please note that it takes a FILE pointer (output of fopen), and a *pointer* to an object of type p2AddrTr. This means that you need to allocate an object of that type. Please note that we are using the term object loosely here, this is not a C++ object, but rather a structure that is typedefed in the byutr.h file. It has several fields, but the most important one is addr which will contain the logical address. The return code lets you know if an address was found or not. Read the byu_tracerreader.c code for further details. You only need to call the NextAddress function. Be sure that you get a recent copy of the BYU header file as explained above in the 32 bit word size section.

Dumping the page table -p

The page table should only be dumped after *all* addresses are processed. You perform a tree traversal, dumping the pages in order. As example, if we ran with a very large page size 2^{22}, or 4 megabytes, we would see the following if we only processed the first 100 addresses:

./pagetable -n 100 -p page_table.txt ~mroch/lib/cs570/trace/trace.sample.tr 10

The contents of page_table.txt should contain all of the valid pages (those assigned to frames) and the first lines should be as follows:

```
00000001 -> 00000000
00000016 -> 0000000b
00000017 -> 00000002
…
```

This means that page 0x1 was assigned to frame 0x0, page 0x16 to frame 0xb, and so on. Note that the pages are in ascending order.

Dumping the logical to physical address translation -t

This is printed to the terminal and can be captured using shell a redirect operator: >. For each address that is executed you will perform the translation, including when an address is seen more than once. Here's the first few lines from the following invocation:

./pagetable -n 100 -t ~mroch/lib/cs570/trace/trace.sample.tr 10

```
0041f760 -> 0001f760
0041f780 -> 0001f780
0041f740 -> 0001f740
```

Reporting the amount of memory used by your page table

This is a summary of the number of bytes that you used in creating your page table and can be obtained by a page table traversal. You need to count the number of bytes for each Level that you allocate as well as the arrays that you allocated to support levels. As an example, suppose your Level implementation uses 24 bytes (use sizeof(Level)) and you allocated an array with 256 entries. If each entry was a pointer (sizeof(Level *)), then you would have used 24 + 256 * sizeof(Level *) bytes for this one level entry alone. You need to compute this throughout the tree and include any supporting structures you used.

Questions about logical to physical address translation

Let us look at the example from above where we translated 0041f760 -> 0001f760 with a single level page table and 4 MB page sizes.

0041f760 can be rewritten in binary as follows:

0000 0000 0100 0001 1111 0111 0110 0000

We had 10 bits of page index, so we mask as follows:

0000 0000 0100 0001 1111 0111 0110 0000

1111 1111 1100 0000 0000 0000 0000 0000 bit-wise and

0000 0000 0100 0000 0000 0000 0000 0000

(We can compute the offset by constructing a 22 bit mask of 1s, resulting in 0x1f760.)

When we right shift by 22 bits we are left with 0000 0000 01. Writing this in groups of four for easy translation to hex, 00 0000 0001, we see that this is page 1. Page 1 mapped to frame 0 (see above), so we take 0 * page size and add the offset of 1f760. Since 0 times anything is still zero, the physical address is 0 + 1f760, or 17f60.

Questions about the data structure

What is a MAP? A MAP is your representation of a page table entry. At a minimum, it is a frame index, but you might choose to include a valid bit like a real page table would. If you do not, you need another way of determining whether or not a frame has been allocated to a page. The last level of your page table will point to an array of type MAP.

What is a frame? A frame is a block of memory that is the same size as a logical page. When you access a logical page, we first find what frame it is associated with and then find how many

bytes we are into the frame. See the question about logical to physical address translation above.

Debugging tip

If you are using gdb, you can print out a number in hexadecimal with p/x; e.g. p/x logical_addr if logical_addr contains a value you want to inspect. I find it much easier to think about bit patterns in hexadecimal, base 10 is a nightmare when you are trying to think about bit positions. Remember printf and cout also have methods to print hex as well.