

Programming Projects

Designing a Virtual Memory Manager

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size $2^{16} = 65,536$ bytes. Your program will read from a file containing logical addresses and, using a TLB as well as a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. The goal behind this project is to simulate the steps involved in translating logical to physical addresses.

Specifics

Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) 8-bit page offset. Hence, the addresses are structured as shown in Figure 9.33.

Other specifics include the following:

- 2^8 entries in the page table
- Page size of 2^8 bytes
- 16 entries in the TLB
- Frame size of 2^8 bytes
- 256 frames
- Physical memory of 65,536 bytes (256 frames \times 256-byte frame size)

Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

Address Translation

Your program will translate logical to physical addresses using a TLB and page table as outlined in Section 8.5. First, the page number is extracted from the logical address, and the TLB is consulted. In the case of a TLB-hit, the frame number is obtained from the TLB. In the case of a TLB-miss, the page table must be consulted. In the latter case, either the frame number is obtained

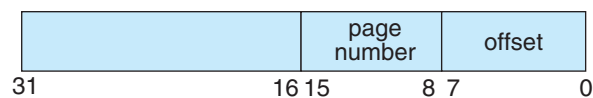


Figure 9.33 Address structure.

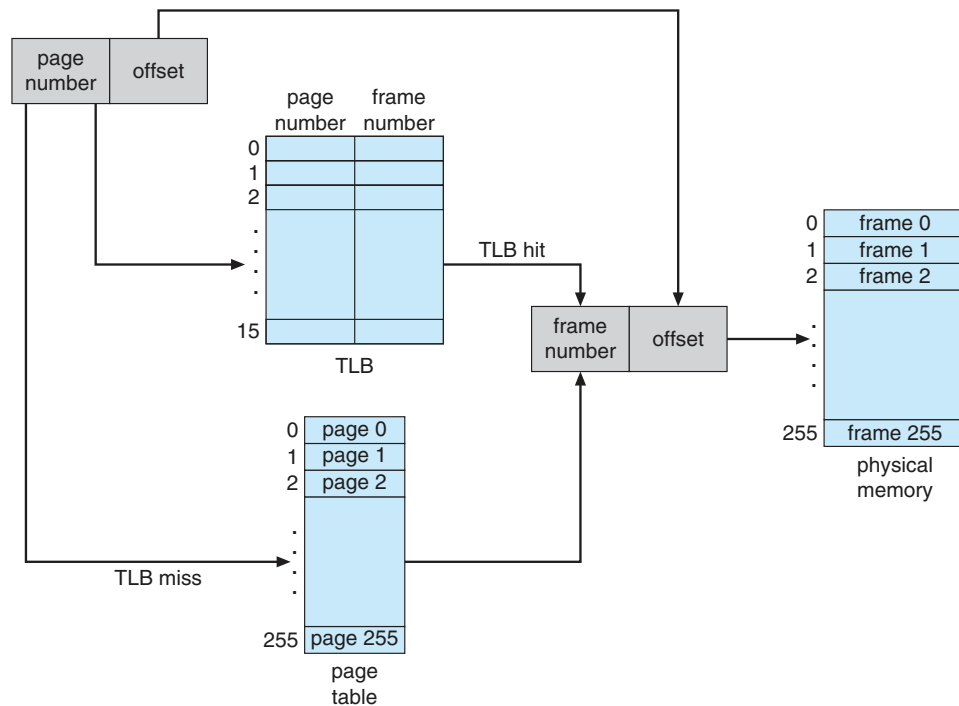


Figure 9.34 A representation of the address-translation process.

from the page table or a page fault occurs. A visual representation of the address-translation process appears in Figure 9.34.

Handling Page Faults

Your program will implement demand paging as described in Section 9.2. The backing store is represented by the file `BACKING_STORE.bin`, a binary file of size 65,536 bytes. When a page fault occurs, you will read in a 256-byte page from the file `BACKING_STORE` and store it in an available page frame in physical memory. For example, if a logical address with page number 15 resulted in a page fault, your program would read in page 15 from `BACKING_STORE` (remember that pages begin at 0 and are 256 bytes in size) and store it in a page frame in physical memory. Once this frame is stored (and the page table and TLB are updated), subsequent accesses to page 15 will be resolved by either the TLB or the page table.

You will need to treat `BACKING_STORE.bin` as a random-access file so that you can randomly seek to certain positions of the file for reading. We suggest using the standard C library functions for performing I/O, including `fopen()`, `fread()`, `fseek()`, and `fclose()`.

The size of physical memory is the same as the size of the virtual address space—65,536 bytes—so you do not need to be concerned about page replacements during a page fault. Later, we describe a modification to this project using a smaller amount of physical memory; at that point, a page-replacement strategy will be required.

Test File

We provide the file `addresses.txt`, which contains integer values representing logical addresses ranging from 0 – 65535 (the size of the virtual address space). Your program will open this file, read each logical address and translate it to its corresponding physical address, and output the value of the signed byte at the physical address.

How to Begin

First, write a simple program that extracts the page number and offset (based on Figure 9.33) from the following integer numbers:

1, 256, 32768, 32769, 128, 65534, 33153

Perhaps the easiest way to do this is by using the operators for bit-masking and bit-shifting. Once you can correctly establish the page number and offset from an integer number, you are ready to begin.

Initially, we suggest that you bypass the TLB and use only a page table. You can integrate the TLB once your page table is working properly. Remember, address translation can work without a TLB; the TLB just makes it faster. When you are ready to implement the TLB, recall that it has only 16 entries, so you will need to use a replacement strategy when you update a full TLB. You may use either a FIFO or an LRU policy for updating your TLB.

How to Run Your Program

Your program should run as follows:

```
./a.out addresses.txt
```

Your program will read in the file `addresses.txt`, which contains 1,000 logical addresses ranging from 0 to 65535. Your program is to translate each logical address to a physical address and determine the contents of the signed byte stored at the correct physical address. (Recall that in the C language, the `char` data type occupies a byte of storage, so we suggest using `char` values.)

Your program is to output the following values:

1. The logical address being translated (the integer value being read from `addresses.txt`).
2. The corresponding physical address (what your program translates the logical address to).
3. The signed byte value stored at the translated physical address.

We also provide the file `correct.txt`, which contains the correct output values for the file `addresses.txt`. You should use this file to determine if your program is correctly translating logical to physical addresses.

Statistics

After completion, your program is to report the following statistics:

1. Page-fault rate—The percentage of address references that resulted in page faults.
2. TLB hit rate—The percentage of address references that were resolved in the TLB.

Since the logical addresses in `addresses.txt` were generated randomly and do not reflect any memory access locality, do not expect to have a high TLB hit rate.

Modifications

This project assumes that physical memory is the same size as the virtual address space. In practice, physical memory is typically much smaller than a virtual address space. A suggested modification is to use a smaller physical address space. We recommend using 128 page frames rather than 256. This change will require modifying your program so that it keeps track of free page frames as well as implementing a page-replacement policy using either FIFO or LRU (Section 9.4).

Bibliographical Notes

Demand paging was first used in the Atlas system, implemented on the Manchester University MUSE computer around 1960 ([Kilburn et al. (1961)]). Another early demand-paging system was MULTICS, implemented on the GE 645 system ([Organick (1972)]). Virtual memory was added to Unix in 1979 [Babaoglu and Joy (1981)]

[Belady et al. (1969)] were the first researchers to observe that the FIFO replacement strategy may produce the anomaly that bears Belady's name. [Mattson et al. (1970)] demonstrated that stack algorithms are not subject to Belady's anomaly.

The optimal replacement algorithm was presented by [Belady (1966)] and was proved to be optimal by [Mattson et al. (1970)]. Belady's optimal algorithm is for a fixed allocation; [Prieve and Fabry (1976)] presented an optimal algorithm for situations in which the allocation can vary.

The enhanced clock algorithm was discussed by [Carr and Hennessy (1981)].

The working-set model was developed by [Denning (1968)]. Discussions concerning the working-set model were presented by [Denning (1980)].

The scheme for monitoring the page-fault rate was developed by [Wulf (1969)], who successfully applied this technique to the Burroughs B5500 computer system.

Buddy system memory allocators were described in [Knowlton (1965)], [Peterson and Norman (1977)], and [Purdom, Jr. and Stigler (1970)]. [Bonwick (1994)] discussed the slab allocator, and [Bonwick and Adams (2001)] extended the discussion to multiple processors. Other memory-fitting algorithms can be found in [Stephenson (1983)], [Bays (1977)], and [Brent (1989)]. A survey of memory-allocation strategies can be found in [Wilson et al. (1995)].

[Solomon and Russinovich (2000)] and [Rusinovich and Solomon (2005)] described how Windows implements virtual memory. [McDougall and Mauro