Virtual Addressing and Paging

Jens Lechtenbörger

June 2020

This text is quite dense. Please do not just read, but *work* through it.

1 Introduction

The presentation accompanying this text is about virtual memory management, which involves the translation of virtual addresses into physical addresses. Physical addresses are what you know from RAM in Hack as `address` input bits, while virtual addresses do not exist in Hack. (Section 2 serves as reminder of RAM in Hack, but it is unlikely to replace your experiences in building RAM chips.)

Virtual addressing provides an additional layer of *abstraction* and *isolation* for physical memory used by processes, threads, and programmers. Oversimplifying a bit, the OS manages each running program as *process*, which you can think of as group of threads that shares one *virtual address space*. Abstraction is provided as at different points in time, the same piece of a process (identified by a single virtual address) may be located at different locations in RAM (identified by different physical addresses) or may not be present in RAM at all. Isolation is provided as different processes have different address spaces, which are protected from each other. So a virtual address, say 42, identifies the same instruction or piece of data for all threads of one process, while 42 points to something else entirely in the context of other processes.

2 Hack Reminder

This section serves as reminder of Hack learning objectives. You may want to repeat CS projects 3 or 4 if in doubt.

Recall that a RAM8 chip consists of 8 registers (each capable of storing one word, which in turn consists of 16 bits) and some addressing logic. In particular, it has an input `address[3]` to address individual words. Note that 3 bits allow to build $2^3 = 8$ binary numbers; thus 3 bits can address exactly 8 registers. Similarly, a RAM16K chip consists of 16384 registers and has an input `address[14]` ($2^{14} = 16384$). In general, for a RAM of size n (where n is a power of 2), $k = log_2 n$ address bits are necessary (and we have $2^k = n$). Such an `address` input is a *physical address* as it determines the number of a physical storage cell (a 16-bit register in Hack).

The Hack architecture uses 15-bit memory addresses and has a RAM of 16384 words (which in turn is embedded into larger memory containing also regions for screen and keyboard). **A**-instructions embed *physical (memory) addresses*;

Hack does not support virtual addressing. In Hack, and in general, a *physical* address is an address that is placed on the memory bus to specify a memory location (the `address` input in Hack HDL files).

Consider the minor variant of the SUM program seen in class, which adds the numbers from 1 to 100, and suppose that it starts at RAM address 0 as shown next. (A real Hack program would be located in ROM; here we suppose the typical von Neumann case of data *and* code in RAM.) Make sure that you understand the comments in that program.

```
// Adds 1+...+100 and stores result in variable sum.
// The first instruction is located in RAM[0].
      @2048 // 2048 is the memory location for loop variable i.
// The second instruction is located in RAM[1].
      M=1   // RAM[2048] = i = 1
// The third instruction is located in RAM[2].
      @2049 // 2049 is the memory location for variable sum.
// The fourth instruction is located in RAM[3].
      M=0   // RAM[2049] = sum = 0
(LOOP) // This label represents the address of the following
       // instruction, which is 4.
// The following instruction is located in RAM[4].
// From now on, RAM locations will be indicated inline.
      @2048 // RAM[4]
      D=M   // RAM[5], D = i
      @100  // RAM[6]
      D=D-A // RAM[7], D = i - 100
      @END  // RAM[8], below we'll see that END is 18, so the
    // assembler produces @18 here.
      D;JGT // RAM[9], if (i-100) > 0 goto END
      @2048 // RAM[10]
      D=M   // RAM[11], D = i
      @2049 // RAM[12]
      M=D+M // RAM[13], sum += i
      @2048 // RAM[14]
      M=M+1 // RAM[15], i++
      @LOOP // RAM[16], as LOOP is really 4, we could have
    // written @4, which is what the assembler will
    // produce anyways.  If that is not obvious, try it!
      0;JMP // RAM[17], goto LOOP
(END) // END represents 18
      @END  // RAM[18], as END is really 18, the assembler
    // produces @18 here.
      0;JMP // RAM[19], infinite loop, jumping back to
    // instruction @18.
// Instructions for other functionality might occur here,
// currently memory is unused until address 2048.
// RAM[2048]: location of variable i
// RAM[2049]: location of variable sum
```

Suppose that you wanted to load several programs into RAM, running several processes and threads (maybe along with round robin scheduling). On disk, each program would embed hard-wired physical memory addresses, e.g., 4, 18, 2048, 2049 for the sample program above. Thus, such a program *must* be loaded to its fixed start address, even if that region is in use by another program while other memory regions are free.

To overcome this shortcoming, from now on we assume that addresses occurring in programs on disk are really virtual addresses.

3 Frames, pages, page tables

The OS does not (in general) allocate individual bytes or words (as this would lead to considerable management overhead), but allocates larger units. This section discusses a common technique called *paging*. With paging, the virtual address space of a process is split into *pages*, while RAM with its physical addresses is split into *frames*. Frames and pages share the *same size*. When a program is to be executed, it needs to be loaded into RAM. Towards that end, the OS allocates frames to the process representing that program and loads (some or all) pages into frames. The data structure to record the mapping from pages to frames is called *page table*.

Given a RAM of 16384 words, RAM could be split into 16 so-called *frames* of 1024 words each, which are numbered from 0 to 15. Thus, `RAM[0]` – `RAM[1023]` are located in frame 0, `RAM[1024]` – `RAM[2047]` are located in frame 1, etc. For example, `RAM[42]` is located in frame 0. In fact, `RAM[42]` really is the 43th word within frame 0. Positions within frames and pages are called *offsets*. For example, `RAM[1042]` is located in frame 1, at offset $1042 - 1024 = 18$.

Suppose that *virtual addresses* are 15 bit wide (leading to addresses from 0 to $2^{15} - 1 = 32767$), and that the virtual address space is split into so-called *pages*, which have the same size as frames, here 1024 words each. Thus, the entire virtual address space covers 32 pages. (You may wonder that the virtual address space is twice as large as the physical RAM. This curious fact does not pose a problem as sketched in Section 5.)

If a process is started, it needs a certain amount of RAM, and the OS allocates memory by allocating entire frames. In the following, suppose again that RAM is used for code and data.

Consider a process P that in total needs 2050 words (similarly to the SUM program shown above, which extends from 0 to 2049). Now suppose that 1150 words are used for code and data (stored in two pages on disk) and 900 words for internal computations (e.g., variables). That process needs three pages (numbered 0, 1, 2) to cover its virtual address space. (Two pages allow for 2048 words, while the process needs 2050 words. As a side remark: Code and data might be allocated in separate RAM areas, so-called segments. For simplicity, assume that they are not separated here.) All memory locations occurring within the code on disk are *virtual* addresses now, so the instruction `sum += i` at virtual address 13 changes the value of virtual address 2049 (which is part of page 2 with its address range 2048 – 3071).

When P starts, the OS needs to transfer the two pages with code and data from disk to frames in RAM and allocate a third frame for the third page. The OS keeps track which frames are free and assigns three to P (assuming that sufficient free frames exist; if they do not exist, page replacement (swapping) is necessary, which will be discussed separately). The assignment from pages to frames is stored in a data structure called *page table*, and every process has its own page table (which is necessary as (a) every process has its own page number 0 and (b) several of those pages numbered 0 may reside in RAM simultaneously). E.g., with several frames allocated to other processes, pages 0 and 1 with code and data of P might be allocated to frames 4 and 5, resp., while page 2 might be allocated to frame 12. The resulting page table is shown in Table 1. (Note that page numbers do not need to be included as they are implicitly given by row numbers starting from 0. Also, later on you will see that page tables contain additional columns with control bits, which is left out for simplicity here.)

Table 1: P's page table

Frame No.
4
5
12

When instructions are executed, the CPU's *memory management unit* (MMU) consults the page table to translate virtual addresses into physical ones. Consider the instruction `sum += i` at virtual address 13 referring to data at virtual address 2049 mentioned above. Virtual address 13 is part of page 0 (ranging from virtual address 0 to 1023) at offset 13. The page table indicates that page 0 is located in frame 4. Frame 4 starts at physical address $4 \cdot 1024$, and so the physical address of the instruction itself is $4 \cdot 1024 + 13 = 4109$, i.e., `RAM[4109]`. Thus, the program counter (PC) has the value 4109 while the instruction is executed. During execution of the instruction, the MMU consults the page table of the process to which the currently running thread belongs to translate virtual address 2049 into a physical one: Virtual address 2049 is located in page 2 at offset 1 ($2049 = 2 \cdot 1024 + 1$), and the page table shows that page 2 is located in frame 12. Thus, the physical address for the necessary piece of data is $12 \cdot 1024 + 1 = 12289$, i.e., `RAM[12289]`.

To sum up, programs make use of stable virtual addresses, while physical addresses are determined flexibly at run-time. Hard-wired physical addresses are avoided.

4 Generalizations

Different processors use different numbers of bits for addresses; typical sizes include 8, 16, 32, 48, 64 bits (note that bit is often abbreviated with "b", while byte as unit of 8 bits is abbreviated with "B"). Importantly, addresses refer to *byte*-sized storage cells (i.e., each cell can store 8 bits), not to words (which is an exceptional case for Hack).

Suppose addresses of 32 bits are used. Then, 2^{32} B = 4 GiB can be addressed. (Ki, Gi, etc. are standardized prefixes for powers of two that are not "too far away" form everyday prefixes K and G for powers of 10.) A typical size for frames and pages might be 2^{12} B = 4 KiB. Please convince yourself that in this situation the 12 least-significant bits (out of all 32 bits) are used to address bytes within pages and frames, i.e., 12 bits determine the offset, while 20 bits remain to enumerate pages and frames. Thus, there are 2^{20} pages (about a million). (In a 32-bit system with less than 4 GiB of RAM, less than 2^{20} frames are available for allocation by the OS, of course. Also, if you do the math for our everyday 64-bit processors, which often use 48-bit addresses, it should be obvious that the size of virtual address spaces is much larger than what we typically have available in terms of RAM.)

As explained above, the page table contains one entry per page (a million entries, per process!), indicating the frame where that page's data is located (if that data is present in RAM at all). Please convince yourself that the translation of a virtual into a physical address is performed by replacing the 20 bits for the page number with the 20 bits for the frame number found in the page table. Note that the 12 offset-bits remain unchanged, as pages and frames have the same size.

5 Locality and Page Faults

The challenge that virtual address spaces may be much larger than physical RAM can be overcome by using *virtual* memory. Briefly, the key insight is a *locality principle* stating that processes (and threads) typically use neighboring pieces of instructions and data over extended periods of time: think of the execution of some method manipulating an object or a loop iterating over an array, for example. In both cases, all memory references can likely be served from two pages: one for code and one for data. Thus, over some period of time, a typical process needs very few of its pages in RAM. Consequently, lots of threads of processes with huge virtual address spaces may be kept in state **Runnable** by the OS with few frames in RAM.

When a threads executes a machine instruction with a virtual address whose page is not present in RAM, a special type of interrupt, called *page fault*, occurs. The handler for that interrupt, the *page fault handler*, is responsible for the transfer of that page from disk to RAM. While the transfer is ongoing, the thread is blocked by the OS.

License Information