What virtual memory *is*

Virtual memory is all about making use of *address space*.

The address space of a processor refers the range of possible addresses that it can use when loading and storing to memory. The address space is limited by the width of the registers, since as we know to load an address we need to issue a `load` instruction with the address to load from stored in a register. For example, registers that are 32 bits wide can hold addresses in a register range from `0x00000000` to `0xFFFFFFF`. 2^32 is equal to 4GB, so a 32 bit processor can load or store to up to 4GB of memory.

64 bit computing

New processors are generally all 64-bit processors, which as the name suggests has registers 64 bits wide. As an exercise, you should work out the address space available to these processors (hint: it's big!).

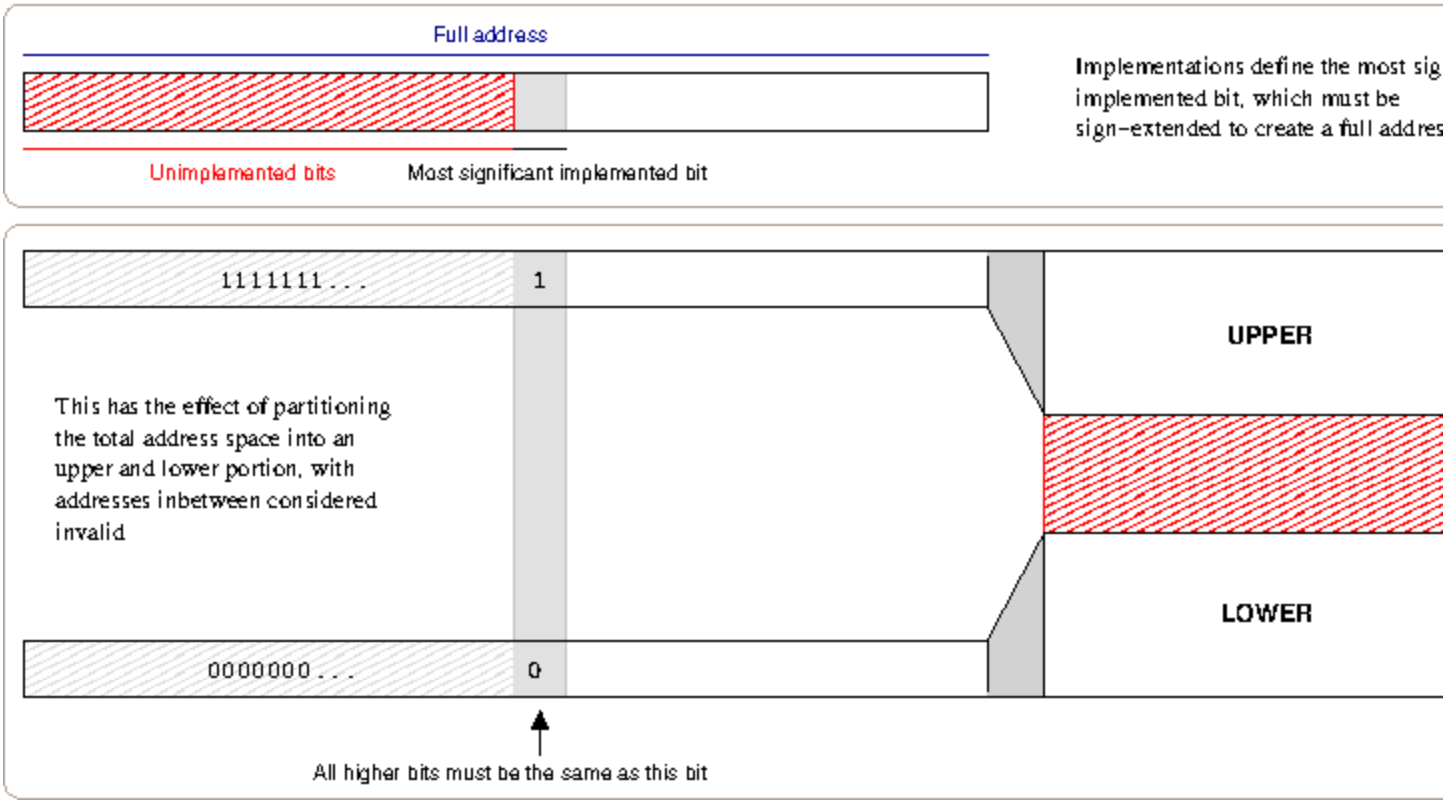
64-bit computing does have some trade-offs against using smaller bit-width processors. Every program compiled in 64-bit mode requires 8-byte pointers, which can increase code and data size, and hence impact both instruction and data cache performance. However, 64-bit processors tend to have more registers, which means less need to save temporary variables to memory when the compiler is under register pressure.

Canonical Addresses

While 64-bit processors have 64-bit wide registers, systems generally do not implement all 64-bits for addressing — it is not actually possible to do `load` or `store` to all 16 exabytes of theoretical physical memory!

Thus most architectures define an *unimplemented* region of the address space which the processor will consider invalid for use. x86-64 and Itanium both define the most-significant valid bit of an address, which must then be sign-extended (see the section called “Sign-extension”) to create a valid address. The result of this is that the total address space is effectively divided into two parts, an upper and a lower portion, with the addresses in-between considered invalid. This is illustrated in Figure 6.1, “Illustration of canonical addresses”. Valid addresses are termed *canonical addresses* (invalid addresses being *non*-canonical).

Figure 6.1. Illustration of canonical addresses



The exact most-significant bit value for the processor can usually be found by querying the processor itself using its informational instructions. Although the exact value is implementation dependent, a typical value would be 48; providing 2^{48} = 256 TiB of usable address-space.

Reducing the possible address-space like this means that significant savings can be made with all parts of the addressing logic in the processor and related components, as they know they will not need to deal with full 64-bit addresses. Since the implementation defines the upper-bits as being signed-extended, this prevents portable operating systems using these bits to store or flag additional information and ensuring compatibility if the implementation wishes to implement more address-space in the future.

Using the address space

As with most components of the operating system, virtual memory acts as an abstraction between the address space and the physical memory available in the system. This means that when a program uses an address that address does not refer to the bits in an actual physical location in memory.

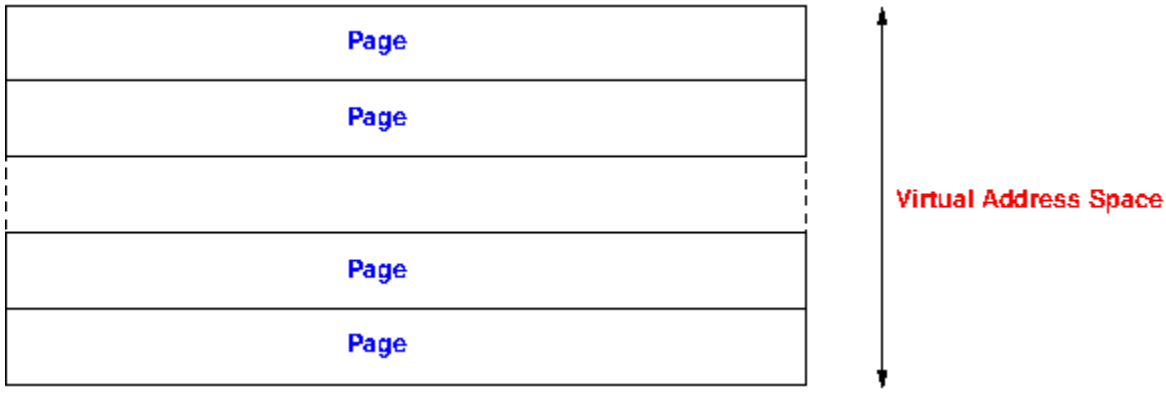
So to this end, we say that all addresses a program uses are *virtual*. The operating system keeps track of virtual addresses and how they are allocated to *physical* addresses. When a program does a load or store from an address, the processor and operating system work together to convert this virtual address to the actual address in the system memory chips.

Pages

The total address-space is divided into individual *pages*. Pages can be many different sizes; generally they are around 4 KiB, but this is not a hard and fast rule and they can be much larger but generally not any smaller. The page is the smallest unit of memory that the operating system and hardware can deal with.

Additionally, each page has a number of attributes set by the operating system. Generally, these include read, write and execute permissions for the current page. For example, the operating system can generally mark the code pages of a process with an executable flag and the processor can choose to not execute any code from pages without this bit set.

Figure 6.2. Virtual memory pages



Programmers may at this point be thinking that they can easily allocate small amounts of memory, much smaller than 4 KiB, using `malloc` or similar calls. This *heap* memory is actually backed by page-size allocations, which the `malloc` implementation divides up and manages for you in an efficient manner.

Physical Memory

Just as the operating system divides the possible address space up into pages, it divides the available physical memory up into *frames*. A frame is just the conventional name for a hunk of physical memory the same size as the system page size.

The operating system keeps a *frame-table* which is a list of all possible pages of physical memory and if they are free (available for allocation) or not. When memory is allocated to a process, it is marked as used in the frame-table. In this way, the operating-system keeps track of all memory allocations.

How does the operating system know what memory is available? This information about where memory is located, how much, attributes and so forth is passed to the operating system by the BIOS during initialisation.

Pages + Frames = Page Tables

It is the job of the operating system is to keep track of which of virtual-page points to which physical frame. This information is kept in a *page-table* which, in its simplest form, could simply be a table where each row contains its associated frame — this is termed a *linear page-table*. If you were to use this simple system, with a 32 bit address-space and 4 KiB pages there would be 1048576 possible pages to keep track of in the page table ($2^{32} \div 4096$); hence the table would be 1048576 entries long to ensure we can always map a virtual page to a physical page.

Page tables can have many different structures and are highly optimised, as the process of finding a page in the page table can be a lengthy process. We will examine page-tables in more depth later.

The page-table for a process is under the exclusive control of the operating system. When a process requests memory, the operating system finds it a free page of physical memory and records the virtual-to-physical translation in the processes page-table. Conversely, when the process gives up memory, the virtual-to-physical record is removed and the underlying frame becomes free for allocation to another process.

Virtual Addresses

When a program accesses memory, it does not know or care where the physical memory backing the address is stored. It knows it is up to the operating system and hardware to work together to map locate the right physical address and thus provide access to the data it wants. Thus we term

the address a program is using to access memory a *virtual address*. A virtual address consists of two parts; the page and an offset into that page.

Page

Since the entire possible address space is divided up into regular sized pages, every possible address resides within a page. The page component of the virtual address acts as an index into the page table. Since the page is the smallest unit of memory allocation within the system there is a trade-off between making pages very small, and thus having very many pages for the operating-system to manage, and making pages larger but potentially wasting memory

Offset

The last bits of the virtual address are called the *offset* which is the location difference between the byte address you want and the start of the page. You require enough bits in the offset to be able to get to any byte in the page. For a 4K page you require (4K == (4 * 1024) == 4096 == 2^{12} ==) 12 bits of offset. Remember that the smallest amount of memory that the operating system or hardware deals with is a page, so each of these 4096 bytes reside within a single page and are dealt with as "one".

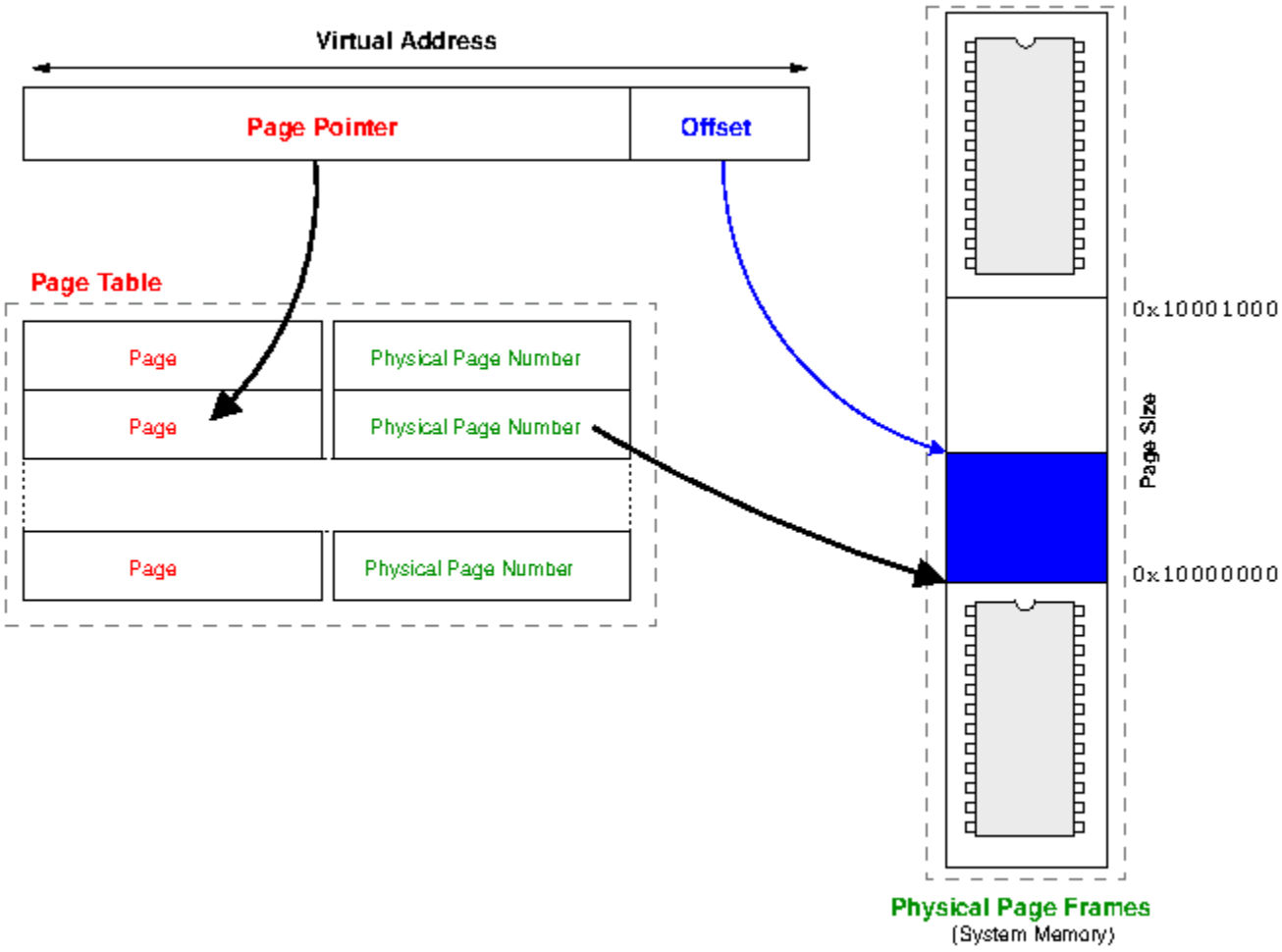
Virtual Address Translation

Virtual address translation refers to the process of finding out which physical page maps to which virtual page.

When translating a virtual-address to a physical-address we only deal with the *page number* . The essence of the procedure is to take the page number of the given address and look it up in the *page-table* to find a pointer to a physical address, to which the offset from the virtual address is added, giving the actual location in system memory.

Since the page-tables are under the control of the operating system, if the virtual-address doesn't exist in the page-table then the operating-system knows the process is trying to access memory that has not been allocated to it and the access will not be allowed.

Figure 6.3. Virtual Address Translation



We can follow this through for our previous example of a simple *linear* page-table. We calculated that a 32-bit address-space would require a table of 1048576 entries when using 4KiB pages. Thus to map a theoretical address of 0x80001234, the first step would be to remove the offset bits. In this case, with 4KiB pages, we know we have 12-bits (2^{12} == 4096) of offset. So we would right-shift out 12-bits of the virtual address, leaving us with 0x80001. Thus (in decimal) the value in row 524289 of the linear page table would be the physical frame corresponding to this page.

You might see a problem with a linear page-table : since every page must be accounted for, whether in use or not, a physically linear page-table is completely impractical with a 64-bit address space. Consider a 64-bit address space divided into (generously large) 64 KiB pages creates $2^{64}/2^{16} = 2^{52}$ pages to be managed; assuming each page requires an 8-byte pointer to a physical location a total of $2^{52}/2^{3} = 2^{49}$ or 512 GiB of contiguous memory is required just for the page table!

Example

Logical Addresses

With a virtual memory system, the main memory can be viewed as a local store for a cache level whose lower level is a disk. Since it is fully associative there is no need for a set field. The address just decomposes into an offset field and a page number field. The number of bits in the offset field is determined by the page size. The remaining bits are the page number.

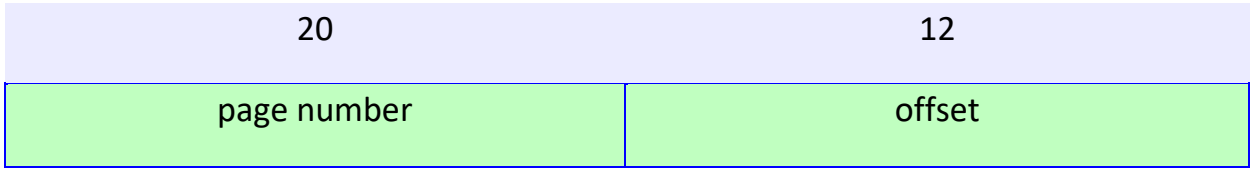
An Example

A computer uses 32-bit byte addressing. The computer uses paged virtual memory with 4KB pages. Calculate the number of bits in the page number and offset fields of a logical address.

Answer

Since there are 4K bytes in a cache block, the offset field must contain 12 bits (2^{12} = 4K). The remaining 20 bits are page number bits.

Thus a logical address is decomposed as shown below.



Page Tables

Virtual memory address translation uses page tables. These are simple arrays in memory indexed by page number. A page table base register (PTBR) holds the base address for the page table of the current process.

Each page table entry contains information about a single page. The most important part of this information is a frame number — where the page is located in physical memory. Address translation combines the frame number with the offset part of a logical address to form a physical address.

The Page Table Base Register (PTBR)

Each process running on a processor needs its own logical address space. This can only be realized if each process has its own page table. To support this, a processor that supports virtual memory must have a page table base register that is accessible by the operating system. For operating system security, this register is only accessible when the processor is in system mode.

The operating system maintains information about each process in a process control block. The page table base address for the process is stored there. The operating system loads this address into the PTBR whenever a process is dispatched.

Page Table Entries

A page table entry contains information about an individual page in a process's logical address space. It typically has a size of 4 bytes (32 bits). It contains the following kinds of information.

- 1 bit indicating if the entry is valid. It is valid only when the page is legal for the process and is in memory. The hardware will trigger an exception if this bit indicates the entry is not valid. The operating system needs another bit to distinguish between legal and illegal accesses.
- 2 or 3 access control bits. These bits can control
    - read access — load instructions
    - write access — store instructions
    - execute access — instruction fetches
- A "dirty" bit to indicate if the page has been modified. A page that has a frame allocated to it but has not been accessed for a while may need to have its frame assigned to a recently accessed page. This is called page replacement. If a page is dirty and it needs to be replaced then the page needs to be written to the disk.
- Bits used by the operating system for approximating how long it has been since the page has been accessed. These bits determine good candidates for replacement. The hardware must provide some support for updating this information. It can be as simple as a single bit that is set whenever a page is accessed.
- Bits to represent the frame number.

When the hardware attempts to access memory and the valid bit is false, the hardware does not need any of the remaining information. It just triggers an exception to bring up the operating system. The operating system can set the valid bit to false for pages that are swapped out to a disk. It can use all of the bits except the valid bit to record the disk location.

With 32 bit physical addresses and 4KB pages the frame number only requires 20 bits. Then a 4B page table entry provides more than enough bits.

Logical Memory Access

A logical memory access (instruction fetch, load, or store) involves two physical memory acesses:

- an access to retrieve the page table entry, and
- an access to fetch an instruction or load or store data.

If nothing is done about it, this doubles the latency for logical memory accesses. All but the very earliest processors that supported virtual memory have used a translation lookaside buffer to eliminate most of the page table entry retrievals.

The Translation Lookaside Buffer

The translation lookaside buffer is just a cache that holds recently accessed page table entries. It can achieve very small miss rates with just a few thousand entries.

Page Table Size

The size of a page table is given by the following equation.

$$\text{page table size} = \frac{\text{logical address space size}}{\text{page size}} \times \text{page table entry size}$$

For example, many processors have 32-bit logical addresses, which results in a 4GB logical address space size. The page table entries size is usually 4B. If the page size is 4KB then the page table size is

$$\text{page table size} = \frac{4\text{GB}}{4\text{KB}} \times 4\text{B} = 4\text{MB}$$

This is excessive, especially on a processor that is running hundreds or even thousands of processes. About 15 years ago, many introductory programming classes had all students running their programs on a mainframe computer. Imagine perhaps 400 students running "Hello, World!" programs, each using 4MB just for page tables.

Most processors use multilevel page tables to reduce the size of page tables for small programs.