13.3. Virtual Memory

The OS's process abstraction provides each process with a virtual memory space. **Virtual memory** is an abstraction that gives each process its own private, logical address space in which its instructions and data are stored. Each process's virtual address space can be thought of as an array of addressable bytes, from address 0 up to some maximum address. For example, on 32-bit systems the maximum address is 2^{32} -1. Processes cannot access the contents of one another's address spaces. Some parts of a process's virtual address space come from the binary executable file it's running (e.g., the *text* portion contains program instructions from the `a.out` file). Other parts of a process's virtual address space are created at runtime (e.g., the *stack*).

Operating systems implement virtual memory as part of the **lone view** abstraction of processes. That is, each process only interacts with memory in terms of its own virtual address space rather than the reality of many processes simultaneously sharing the computer's physical memory (RAM). The OS also uses its virtual memory implementation to protect processes from accessing one another's memory spaces. As an example, consider the following simple C program:

```c
/* a simple program */
#include <stdio.h>

int main(int argc, char* argv[]) {
    int x, y;

    printf("enter a value: ");
    scanf("%d", &y);

    if (y > 10) {
        x = y;
    } else {
        x = 6;
    }
    printf("x is %d\n", x);

    return 0;
}
```

If two processes simultaneously execute this program, they each get their own copy of stack memory as part of their separate virtual address spaces. As a result, if one process executes `x = 6` it will have

no effect on the value of `x` in the other process — each process has its own copy of `x`, in its private virtual address space, as shown in Figure 1.

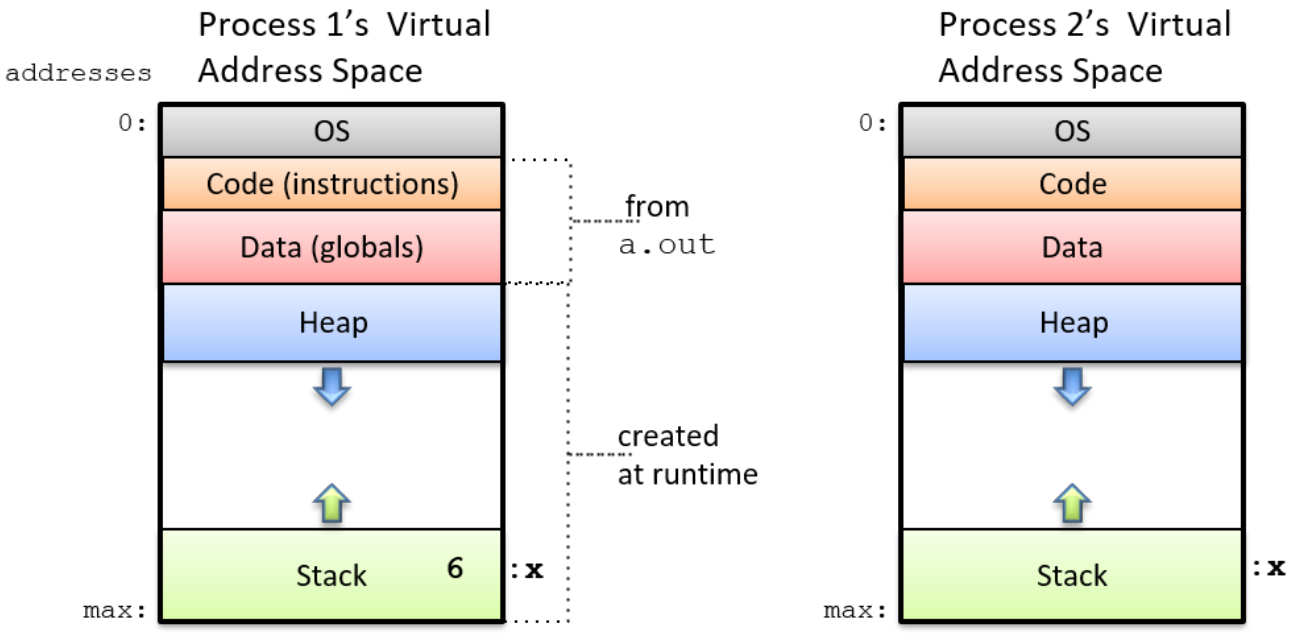

Figure 1. Two executions of a.out results in two processes, each running isolated instances of the a.out program. Each process has its own private virtual address space, containing its copies of program instructions, global variables, and stack and heap memory space. For example, each may have a local variable x in the stack portion of their virtual address spaces.

A process's virtual address space is divided into several sections, each of which stores a different part of the process's memory. The top part (at the lowest addresses) is reserved for the OS and can only be accessed in kernel mode. The text and data parts of a process's virtual address space are initialized from the program executable file (`a.out`). The text section contains the program instructions, and the data section contains global variables (the data portion is actually divided into two parts, one for initialized global variables and the other for uninitialized globals).

The stack and heap sections of a process's virtual address space vary in size as the process runs. Stack space grows in response to the process making function calls, and shrinks as it returns from functions. Heap space grows when the process dynamically allocates memory space (via calls to `malloc`), and shrinks when the process frees dynamically allocated memory space (via calls to `free`). The heap and stack portions of a process's memory are typically located far apart in its address space to maximize the amount of space either can use. Typically, the stack is located at the bottom of a process's address space (near the maximum address), and grows upward into lower addresses as stack frames are added to the top of the stack in response to a function call.

About Heap and Stack Memory

The actual total capacity of heap and stack memory space does not typically change on every call to `malloc` and `free`, nor on every function call and return. Instead, these actions often only make changes to how much of the currently allocated heap and stack parts of the virtual memory space are actively being used by the process. Sometimes, however, these actions do result in changes to the total capacity of the heap or stack space.

The operating system is responsible for managing a process's virtual address space, including changing the total capacity of heap and stack space. The system calls `brk`, `sbrk`, or `mmap` can be used to request that the OS change the total capacity of heap memory. C programmers do not usually invoke these system calls directly. Instead, C programmers call the standard C library function `malloc` (and `free`) to allocate (and free) heap memory space. Internally, the standard C library's user-level heap manager may invoke one of these system calls to request that the OS change the size of heap memory space to satisfy a `malloc` request.

13.3.1. Memory Addresses

Because processes operate within their own virtual address spaces, operating systems must make an important distinction between two types of memory addresses. **Virtual addresses** refer to storage locations in a process's virtual address space, and **physical addresses** refer to storage locations in physical memory (RAM).

Physical Memory (RAM) and Physical Memory Addresses

From the Storage and Memory Hierarchy Chapter, we know that physical memory (RAM) can be viewed as an array of addressable bytes in which addresses range from 0 to a maximum address value based on the total size of RAM. For example, in a system with 2 gigabytes (GB) of RAM, physical memory addresses range from 0 to 2^{31} -1 (1 GB is 2^{30} bytes, so 2 GB is 2^{31} bytes).

In order for the CPU to run a program, the program's instructions and data must be loaded into RAM by the OS; the CPU cannot directly access other storage devices (e.g., disks). The OS manages RAM and determines which locations in RAM should store the virtual address space contents of a process. For example, if two processes, P1 and P2, run the example program listed above, then P1 and P2 have separate copies of the `x` variable, each stored at a different location in RAM. That is, P1's `x` and P2's `x` have different physical addresses. If the OS gave P1 and P2 the same physical address for their `x` variables, then P1 setting `x` to 6 would also modify P2's value of `x`, violating the per-process private virtual address space.

At any point in time, the OS stores in RAM the address space contents from many processes as well as OS code that it may map into every process's virtual address space (OS code is typically loaded starting at address 0x0 of RAM). Figure 2 shows an example of the OS and three processes (P1, P2, and P3) loaded into RAM. Each process gets its own separate physical storage locations for its address space

contents (e.g., even if P1 and P2 run the same program, they get separate physical storage locations for their variable `x`).

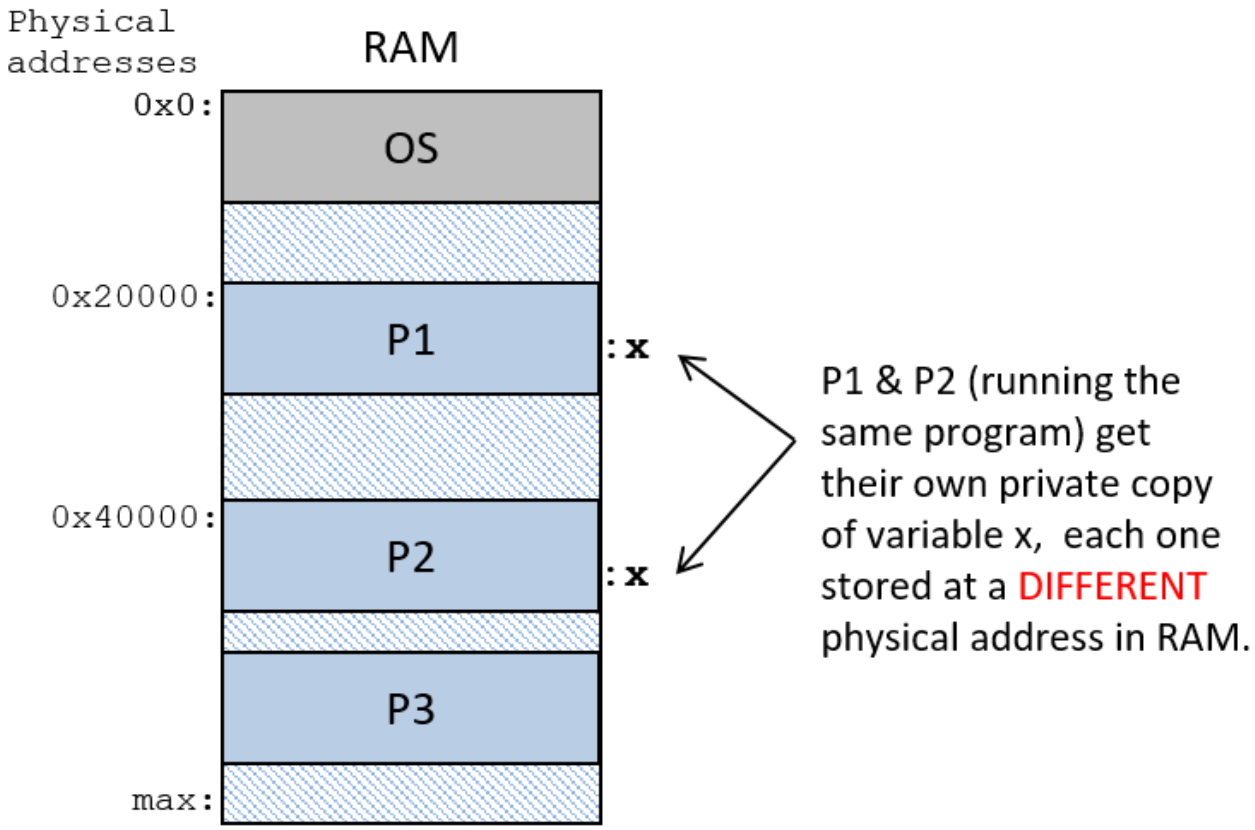

Figure 2. Example RAM contents showing OS loaded at address 0x0, and processes loaded at different physical memory addresses in RAM. If P1 and P2 are running the same a.out, P1's physical address for x is different from P2's physical address for x.

Virtual Memory and Virtual Addresses

Virtual memory is the per-process view of its memory space, and **virtual addresses** are addresses in the process's view of its memory. If two process run the same binary executable, they have exactly the same virtual addresses for function code and for global variables in their address spaces (the virtual addresses of dynamically allocated space in heap memory and of local variables on the stack may vary slightly between the two processes due to runtime differences in their two separate executions). In other words, both processes will have the same virtual addresses for the location of their `main` function, and the same virtual address for the location of a global variable `x` in their address spaces, as shown in Figure 3.

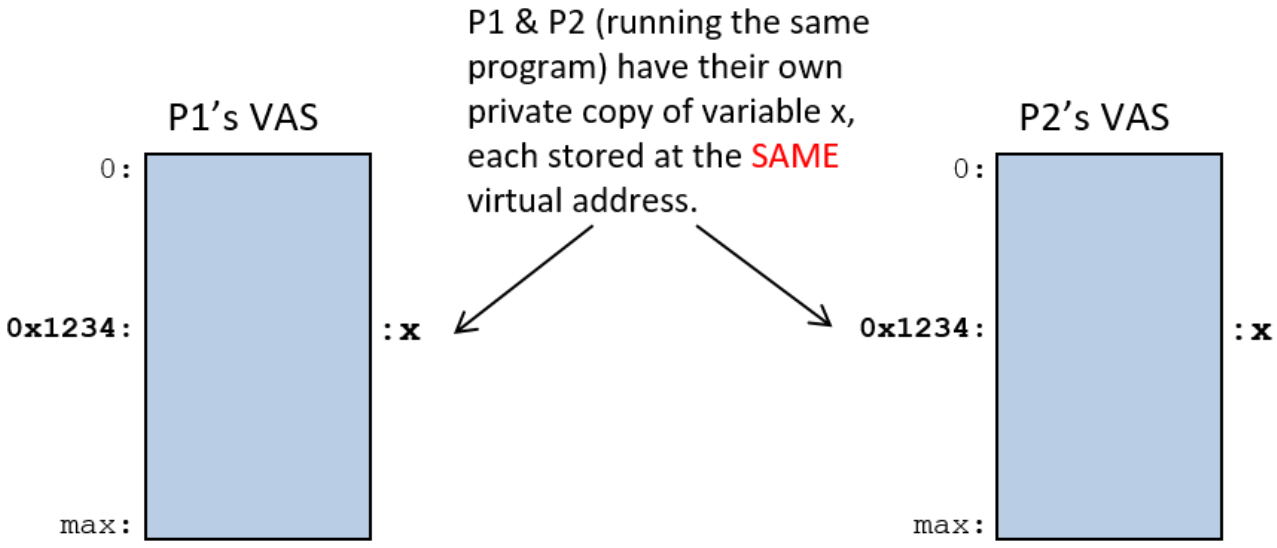

Figure 3. Example virtual memory contents for two processes running the same a.out file. P1 and P2 have the same virtual address for global variable x.

13.3.2. Virtual Address to Physical Address Translation

A program's assembly and machine code instructions refer to virtual addresses. As a result, if two processes execute the same `a.out` program, the CPU executes instructions with identical virtual addresses to access corresponding parts of their two separate virtual address spaces. For example, suppose that `x` is at virtual address 0x24100, then assembly instructions to set `x` to 6 might look like this:

```
movl $0x24100, %eax     # load 0x24100 into register eax
movl $6, (%eax)         # store 6 at memory address 0x24100
```

At runtime the OS loads each of the processes' `x` variables at different physical memory addresses (at different locations in RAM). This means that whenever the CPU executes a load or store instruction to memory that specify virtual addresses, the virtual address from the CPU must be translated to its corresponding physical address in RAM before reading or writing the bytes from RAM.

Because virtual memory is an important and core abstraction implemented by operating systems, processors generally provide some hardware support for virtual memory. An OS can make use of this hardware-level virtual memory support to perform virtual to physical address translations quickly, avoiding having to trap to the OS to handle every address translation. A particular OS chooses how much of the hardware support for paging it uses in its implementation of virtual memory. There is often a trade-off in speed versus flexibility when choosing a hardware-implemented feature versus a software-implemented feature.

The **memory management unit** (MMU) is the part of the computer hardware that implements address translation. Together, the MMU hardware and the OS translate virtual to physical addresses when applications access memory. The particular hardware/software split depends on the specific combination of

hardware and OS. At its most complete, MMU hardware performs the full translation: it takes a virtual address from the CPU and translates it to a physical address that is used to address RAM (as shown in Figure 4). Regardless of the extent of hardware support for virtual memory, there will be some virtual-to-physical translations that the OS has to handle. In our discussion of virtual memory, we assume a more complete MMU that minimizes the amount of OS involvement required for address translation.

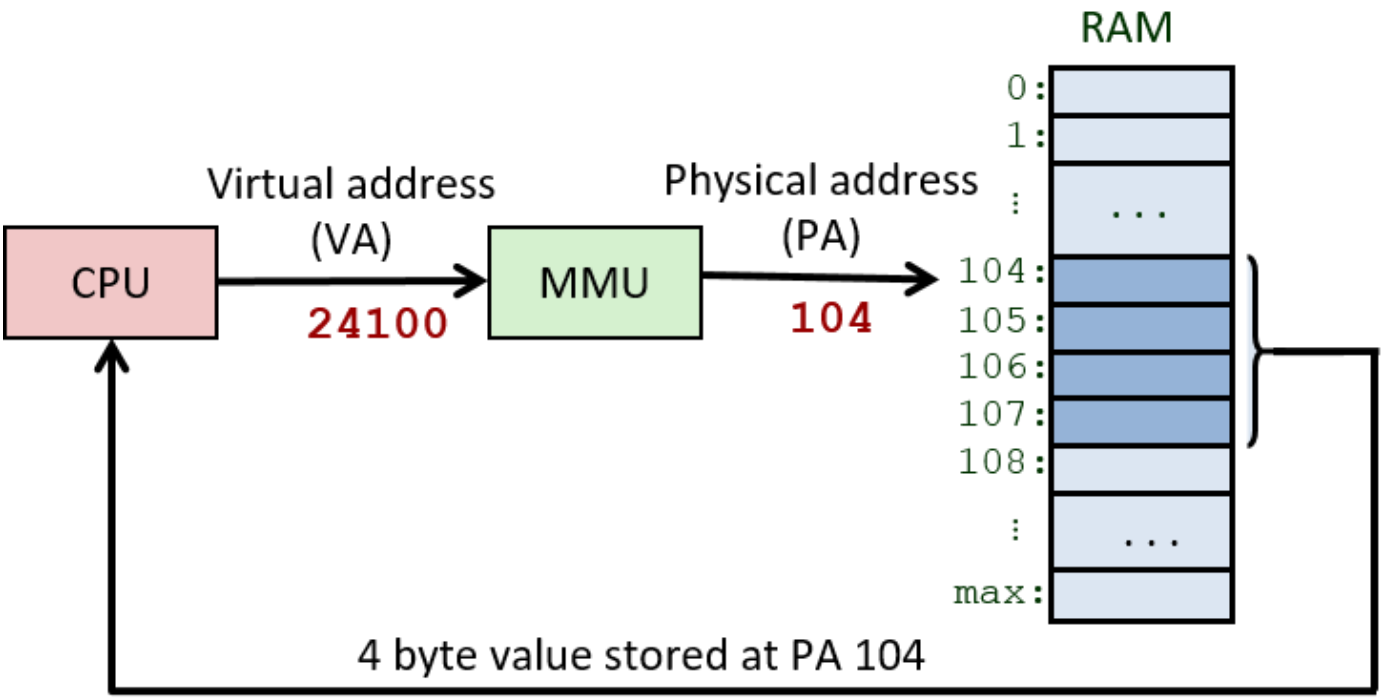

Figure 4. The memory management unit (MMU) maps virtual to physical addresses. Virtual addresses are used in instructions executed by the CPU. When the CPU needs to fetch data from physical memory, the virtual address is first translated by the MMU to a physical addresses that is used to address RAM.

The OS maintains virtual memory mappings for each process to ensure that it can correctly translate virtual to physical addresses for any process that runs on the CPU. During a context switch, the OS updates the MMU hardware to refer to the swapped-on process's virtual to physical memory mappings. The OS protects processes from accessing one another's memory spaces by swapping the per-process address mapping state on a context switch — swapping the mappings on a context switch ensures that one process's virtual addresses will not map to physical addresses storing another process's virtual address space.

13.3.3. Paging

Although many virtual memory systems have been proposed over the years, paging is now the most widely used implementation of virtual memory. In a **paged virtual memory** system, the OS divides the virtual address space of each process into fixed-sized chunks called **pages**. The OS defines the page size for the system. Page sizes of a few kilobytes are commonly used in general-purpose operating systems today — 4 KB (4,096 bytes) is the default page size on many systems.

Physical memory is similarly divided by the OS into page-sized chunks called **frames**. Because pages and frames are defined to be the same size, any page of a process's virtual memory can be stored in any frame of physical RAM.

In a paging system:

- Pages and frames are the same size, so any page of virtual memory can be loaded into (stored) in any physical frame of RAM.
- A process's pages do not need to be stored in contiguous RAM frames (at a sequence of addresses all next to one another in RAM).
- Not every page of virtual address space needs to be loaded into RAM for a process to run.

Figure 5 shows an example of how pages from a process's virtual address space may map to frames of physical RAM.

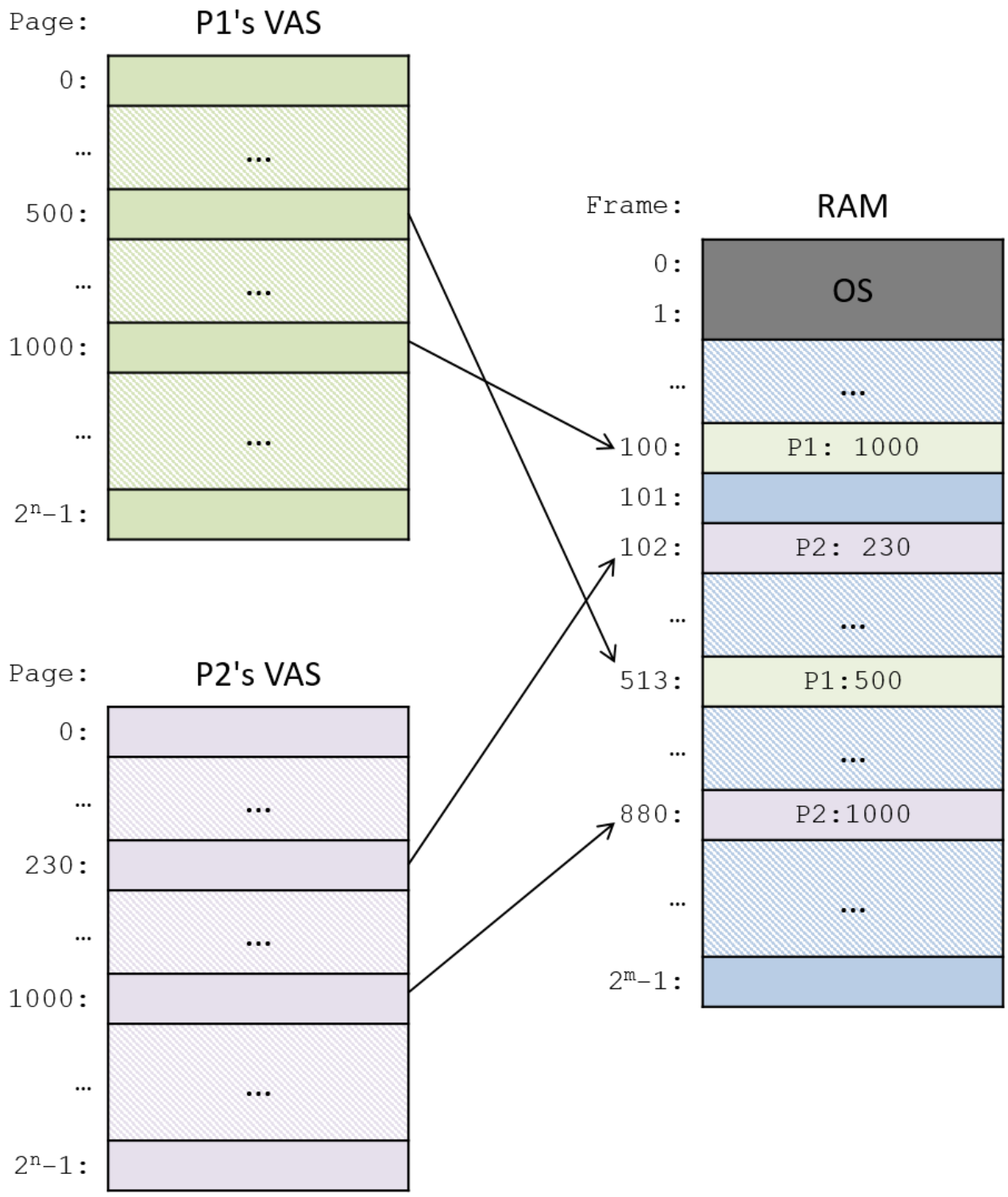

Figure 5. Paged virtual memory. Individual pages of a process's virtual address space are stored in RAM frames. Any page of virtual address space can be loaded into (stored at) any frame of physical memory. In this example, P1's virtual page 1000 is stored in physical frame 100, and its page 500 resides in frame 513. P2's virtual page 1000 is stored in physical frame 880, and its page 230 resides in frame 102.

Virtual and Physical Addresses in Paged Systems

Paged virtual memory systems divide the bits of a virtual address into two parts: the high-order bits specify the **page number** on which the virtual address is stored, and the low-order bits correspond to the **byte offset** within the page (which byte from the top of the page corresponds to the address).

Similarly, paging systems divide physical addresses into two parts: the high-order bits specify the **frame number** of physical memory, and the low-order bits specify the **byte offset** within the frame. Because frames and pages are the same size, the byte offset bits in a virtual address are identical to the byte offset bits in its translated physical address. Virtual addresses differ from their translated physical addresses in their high-order bits, which specify the virtual page number and physical frame number.

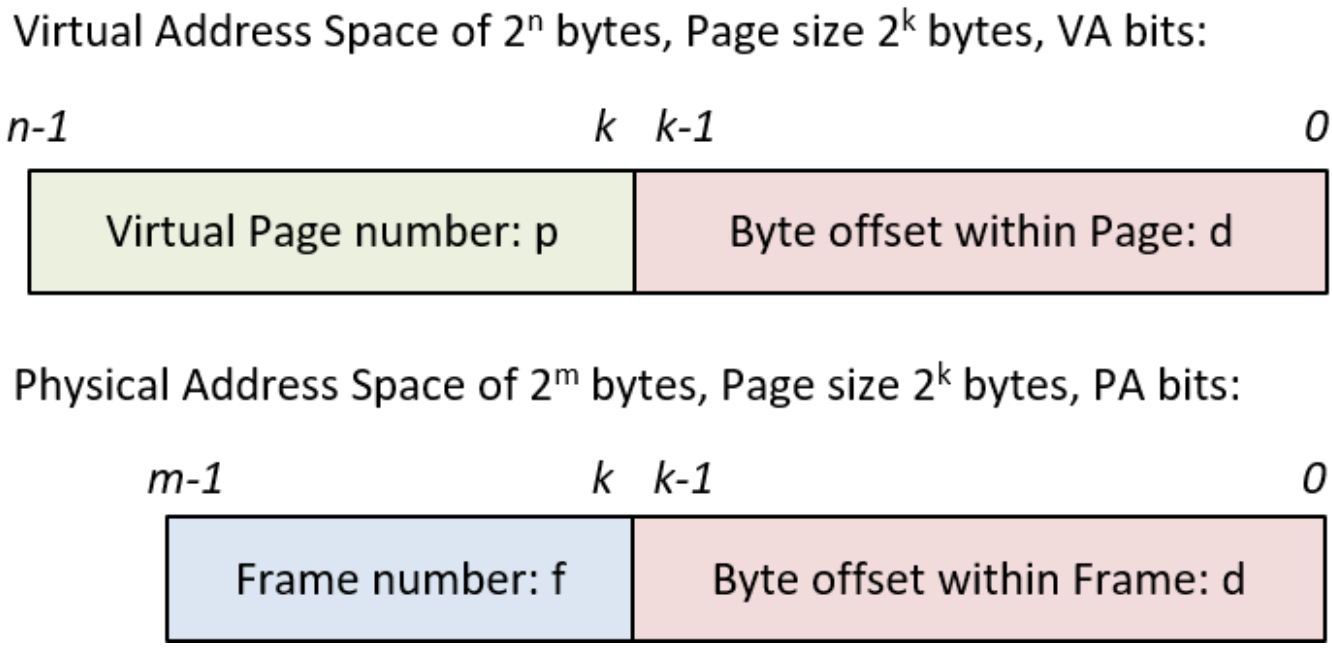

Figure 6. The address bits in virtual and physical addresses

For example, consider a (very tiny) system with 16-bit virtual addresses, 14-bit physical addresses, and 8-byte pages. Because the page size is eight bytes, the low-order three bits of physical and virtual addresses define the byte offset into a page or frame — three bits can encode eight distinct byte offset values, 0-7 (2^3 is 8). This leaves the high-order 13 bits of the virtual address for specifying the page number and the high-order 11 bits of the physical address for specifying frame number, as shown in the example in Figure 7.

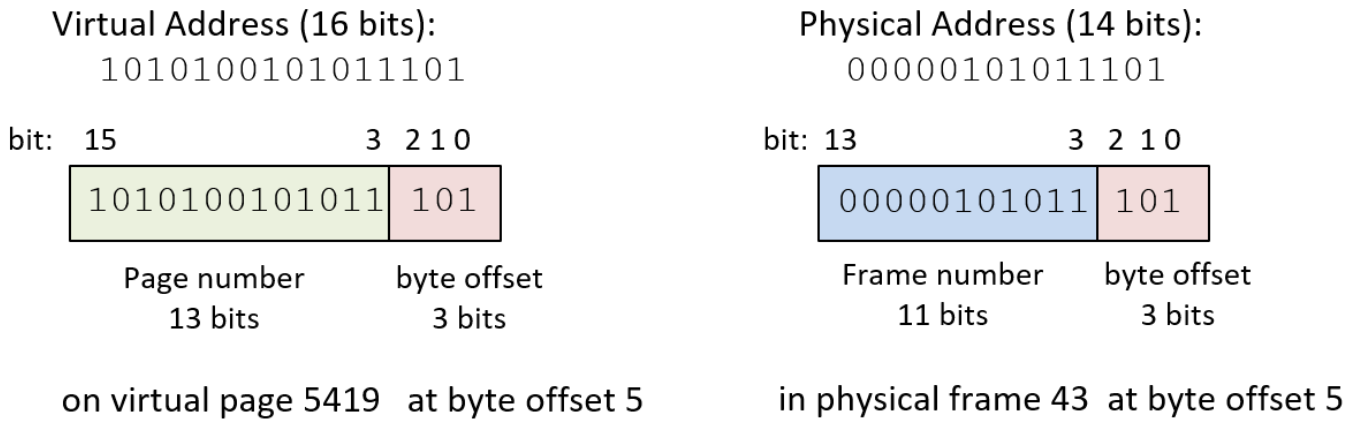

Figure 7. Virtual and physical address bit divisions in an example system with 16-bit virtual addresses, 14-bit physical addresses, and a page size of 8 bytes.

In the example in Figure 7, virtual address 43357 (in decimal) has a byte offset of 5 (0b101 in binary), the low-order 3 bits of the address, and a page number of 5419 (0b1010100101011), the high-order 13 bits of the address. This means that the virtual address is at byte 5 from the top of page 5419.

If this page of virtual memory is loaded into frame 43 (0b00000101011) of physical memory, then its physical address is 349 (0b00000101011101), where the low-order 3 bits (0b101) specify the byte offset, and the high-order 11 bits (0b00000101011) specify the frame number. This means that the physical address is at byte 5 from the top of frame 43 of RAM.

Page Tables for Virtual-to-Physical Page Mapping

Because every page of a process's virtual memory space can map to a different frame of RAM, the OS must maintain mappings for every virtual page in the process's address space. The OS keeps a per-process **page table** that it uses to store the process's virtual page number to physical frame number mappings. The page table is a data structure implemented by the OS that is stored in RAM. Figure 8 shows an example of how the OS may store two process's page tables in RAM. The page table of each process stores the mappings of its virtual pages to their physical frames in RAM such that any pages of virtual memory can be stored in any physical frame of RAM.

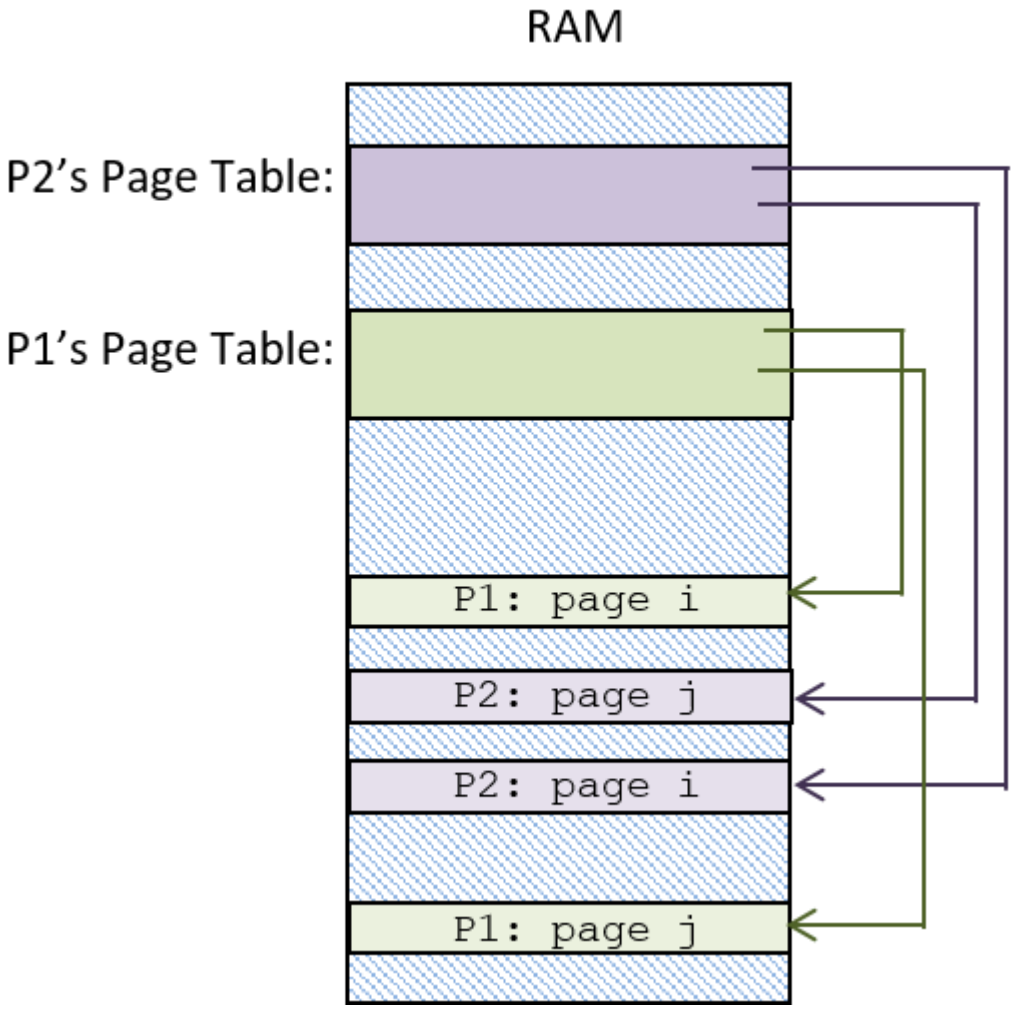

Figure 8. Every process has a page table containing its virtual page to physical frame mappings. Page tables, stored in RAM, are used by the system to translate process's virtual addresses to physical addresses that are used to address locations in RAM. This example shows the separate page tables stored in RAM for processes P1 and P2, each page table with its own virtual page to physical frame mappings.

For each page of virtual memory, the page table stores one **page table entry** (PTE) that contains the frame number of physical memory (RAM) storing the virtual page. A PTE may also contain other information about the virtual page, including a **valid bit** that is used to indicate whether the PTE stores a valid mapping. If a page's valid bit is zero, then the page of the process's virtual address space is not currently loaded into physical memory.

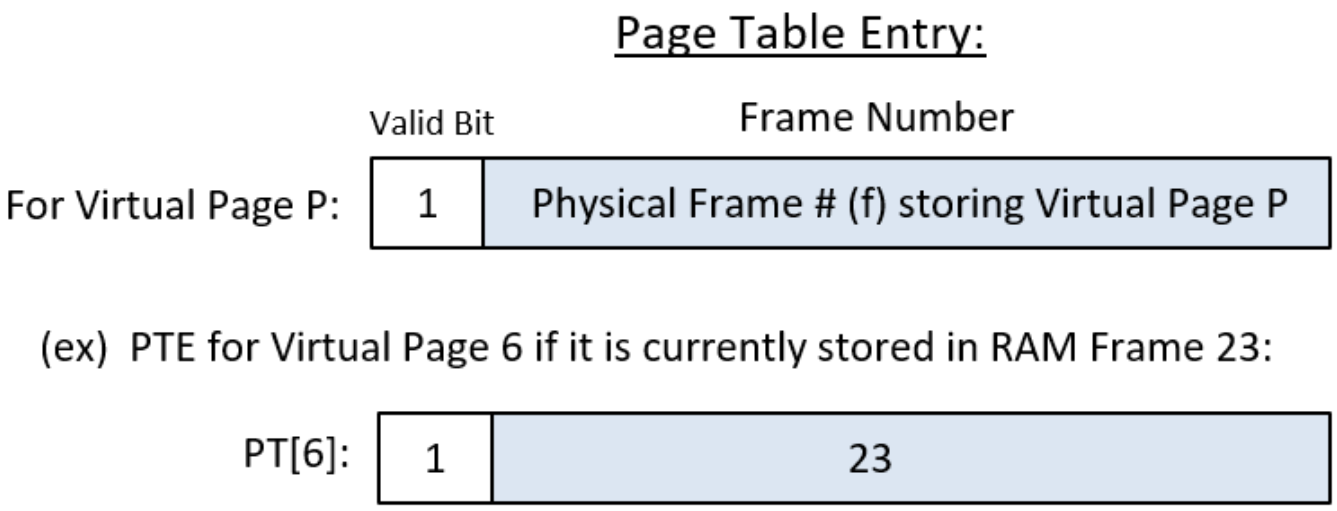

Figure 9. A page table entry (PTE) stores the frame number (23) of the frame of RAM in which the virtual page is loaded. We list the frame number (23) in decimal, although it is really encoded in binary in the PTE entry (0…010111). A valid bit of 1 indicates that this entry stores a valid mapping.

Using a Page Table to Map Virtual to Physical Addresses

There are four steps to translating a virtual address to a physical address (shown in Figure 10). The particular C13-OS/hardware combination determines which of the OS or the hardware performs all or part of each step. We assume a full-featured MMU that performs as much of the address translation as possible in hardware in describing these steps, but on some systems the OS may perform parts of these steps.

1. First, the MMU divides the bits of the virtual address into two parts: for a page size of 2^k bytes, the low-order k bits (VA bits (k-1) to 0) encode the byte offset (d) into the page, and the high-order n-k bits (VA bits (n-1) to k) encode the virtual page number (p).
2. Next, the page number value (p) is used by the MMU as an index into the page table to access the PTE for page p. Most architectures have a **page table base register** (PTBR) that stores the RAM address of the running process's page table. The value in the PTBR is combined with the page number value (p) to compute the address of the PTE for page p.
3. If the valid bit in the PTE is set (is 1), then the frame number in the PTE represents a valid VA to PA mapping. If the valid bit is 0, then a page fault occurs, triggering the OS to handle this address translation (we discuss the OS page fault handling later).
4. The MMU constructs the physical address using the frame number (f) bits from the PTE entry as the high-order bits, and the page offset (d) bits from the VA as the low-order bits of the physical address.

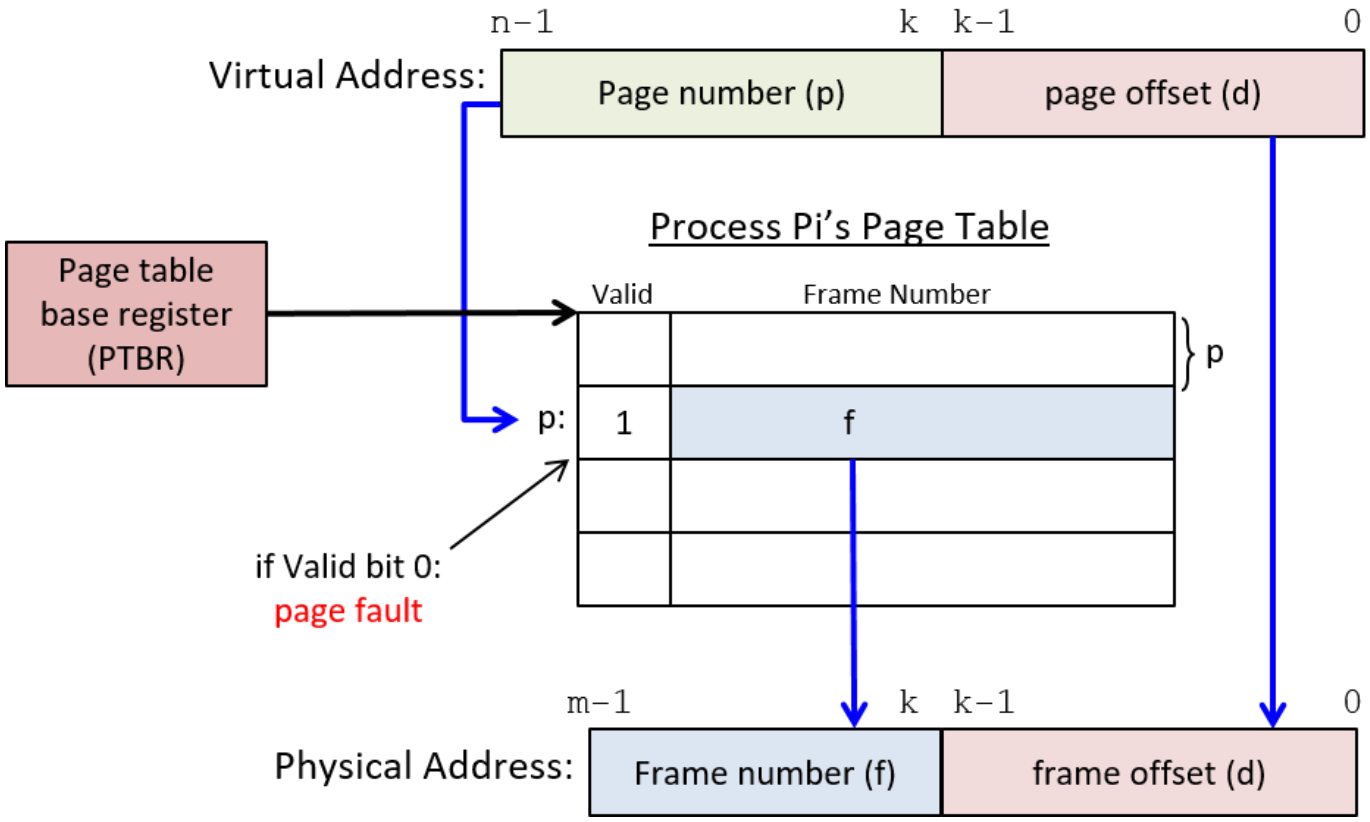

Figure 10. A process's page table is used to perform virtual to physical address translations. The PTBR stores the base address of the currently running process's page table.

An Example: Mapping VA to PA with a Page Table

Consider an example (tiny) paging system where:

- the page size is 4 bytes
- the virtual addresses are 6 bits (the high-order 4 bits are the page number and the low-order 2 bits are the byte offset)
- the physical addresses are 7 bits

Assume that the page table for process P1 in this system looks like Table 1 (values are listed in both decimal and binary).

Table 1. Process P1's Page Table

Entry	Valid	Frame #
0 (0b0000)	1	23 (0b10111)
1 (0b0001)	0	17 (0b10001)
2 (0b0010)	1	11 (0b01011)

Entry	Valid	Frame #
3 (0b0011)	1	16 (0b10000)
4 (0b0100)	0	08 (0b01000)
5 (0b0101)	1	14 (0b01110)
…	…	…
15 (0b1111)	1	30 (0b11110)

Using the information provided in this example suggests several important things about address sizes, parts of addresses, and address translation, including:

- The size of (number of entries in) the page table is determined by the number of bits in the virtual address and the page size in the system. The high-order 4 bits of each 6-bit virtual address specifies the page number, so there are 16 (2^4) total pages of virtual memory. Since the page table has one entry for each virtual page, there are a total of 16 page table entries in each process's page table.
- The size of each page table entry (PTE) depends on the number of bits in the physical address and the page size in the system. Each PTE stores a valid bit and a physical frame number. The valid bit requires a single bit. The frame number requires 5 bits because physical addresses are 7 bits and the page offset is the low-order 2 bits (to address the 4 bytes on each page), which leaves the 5 high-order bits for the frame number. Thus, each PTE entry requires 6 bits: 1 for the valid bit, and 5 for the frame number.
- The maximum sizes of virtual and physical memory are determined by the number of bits in the addresses. Because virtual addresses are 6 bits, 2^6 bytes of memory can be addressed, so each process's virtual address space is 2^6 (or 64) bytes. Similarly, the maximum size of physical memory is 2^7 (or 128) bytes.
- The page size, the number of bits in virtual and physical addresses, and the page table determine the mapping of virtual to physical addresses. For example, if process P1 executes an instruction to load a value from its virtual address 0b001110, its page table is used to convert the virtual address to physical address 0b1000010, which is then used to access the value in RAM.

The virtual address (VA) to physical address (PA) translation steps are:

1. Separate the VA bits into the page number (p) and byte offset (d) bits: the high-order four bits are the page number (0b0011 or page 3) and the lower-order two bits are the byte offset into the page (0b10 or byte 2).

2. Use the page number (3) as an index into the page table to read the PTE for virtual page 3 (PT[3]: valid:1 frame#:16).
3. Check the valid bit for a valid PTE mapping. In this case, the valid bit is 1, so the PTE contains a valid mapping, meaning that virtual memory page 3 is stored in physical memory frame 16.
4. Construct the physical address using the five-bit frame number from the PTE as the high-order address bits (0b10000), and the low-order two-bit offset from the virtual address (0b10) as the lower-order two bits: the physical address is 0b1000010 (in RAM frame 16 at byte offset 2).

Paging Implementation

Most computer hardware provides some support for paged virtual memory, and together the OS and hardware implement paging on a given system. At a minimum, most architectures provide a page table base register (PTBR) that stores the base address of the currently running process's page table. To perform virtual-to-physical address translations, the virtual page number part of a virtual address is combined with the value stored in the PTBR to find the PTE entry for the virtual page. In other words, the virtual page number is an index into the process's page table, and its value combined with the PTBR value gives the RAM address of the PTE for page p (e.g., PTBR + p × (PTE size) is the RAM address of the PTE for page p). Some architectures may support the full page table lookup by manipulating PTE bits in hardware. If not, then the OS needs to be interrupted to handle some parts of page table lookup and accessing the PTE bits to translate a virtual address to a physical address.

On a context switch, the OS *saves and restores* the PTBR values of processes to ensure that when a process runs on the CPU it accesses its own virtual-to-physical address mappings from its own page table in RAM. This is one mechanism through which the OS protects processes' virtual address spaces from one another; changing the PTBR value on context switch ensures that a process cannot access the VA-PA mappings of another process, and thus it cannot read or write values at physical addresses that store the virtual address space contents of any other processes.

An Example: Virtual-to-Physical Address Mappings of Two Processes

As an example, consider an example system with eight-byte pages, seven-bit virtual addresses, and six-bit physical addresses.

Table 2. Example Process Page Tables

P1's Page Table			**P2's Page Table**		
Entry	Valid	Frame #	Entry	Valid	Frame #
0	1	3	0	1	1
1	1	2	1	1	4

P1's Page Table			P2's Page Table		
2	1	6	2	1	5
…			…		
11	1	7	11	0	3
…			…		

Given the following current state of the (partially shown) page tables of two processes (P1 and P2) in Table 2, let's compute the physical addresses for the following sequence of virtual memory addresses generated from the CPU (each address is prefixed by the process that is running on the CPU):

```
P1: 0000100
P1: 0000000
P1: 0010000
              <---- context switch
P2: 0010000
P2: 0001010
P2: 1011001
              <---- context switch
P1: 1011001
```

First, determine the division of bits in virtual and physical addresses. Since the page size is eight bytes, the three low-order bits of every address encodes the page offset (d). Virtual addresses are seven bits. Thus, with three bits for the page offset, this leaves the four high-order bits for specifying the page number (p). Since physical addresses are six bits long and the low-order three are for the page offset, the high-order three bits specify the frame number.

Next, for each virtual address, use its page number bits (p) to look up in the process's page table the PTE for page p. If the valid bit in the PTE is set, then use the frame number (f) for the high-order bits of the PA. The low-order bits of the PA come from the byte-offset bits (d) of the VA.

The results are shown in Table 3 (note which page table is being used for the translation of each address).

Table 3. Address Mappings for the Example Sequence of Memory Accesses from Processes P1 and P2. Note that a Context Switch Changes which Page Table is used for Address Translation.

Process	VirtAddr	p	d	PTE	f	d	PhysAddr

Process	VirtAddr	p	d	PTE	f	d	PhysAddr
P1	0000100	0000	100	PT[0]: 1(v),3(f)	011	100	011100
P1	0000000	0000	000	PT[0]: 1(v),3(f)	011	000	011000
P1	0010000	0010	000	PT[2]: 1(v),6(f)	110	000	110000
Context Switch P1 to P2							
P2	0010000	0010	000	PT[2]: 1(v),5(f)	101	000	101000
P2	0001010	0001	010	PT[1]: 1(v),4(f)	100	010	100010
P2	1011001	1011	001	PT[11]: 0(v),3(f)	**page fault (valid bit 0)**		
Context Switch P2 to P1							
P1	1011001	1011	001	PT[11]: 1(v),7(f)	111	001	111001

As one example, consider the first address accesses by process P1. When P1 accesses its virtual address 8 (0b0000100), the address is divided into its page number 0 (0b0000) and its byte offset 4 (0b100). The page number, 0, is used to look up PTE entry 0, whose valid bit is 1, indicating a valid page mapping entry, and whose frame number is 3 (0b011). The physical address (0b011100) is constructed using the frame number (0b011) as the high-order bits and the page offset (0b100) as the low-order bits.

When process P2 is context switched on the CPU, its page table mappings are used (note the different physical addresses when P1 and P2 access the same virtual address 0b0010000). When P2 accesses a PTE entry with a 0 valid bit, it triggers a page fault to the OS to handle.

13.3.4. Memory Efficiency

One of the primary goals of the operating system is to efficiently manage hardware resources. System performance is particularly dependent on how the OS manages the memory hierarchy. For example, if a process accesses data that are stored in RAM, the process will run much faster than if those data are on disk.

The OS strives to increase the degree of multiprogramming in the system in order to keep the CPU busy doing real work while some processes are blocked waiting for an event like disk I/O. However, because RAM is fixed-size storage, the OS must make decisions about which process to load in RAM at any point in time, possibly limiting the degree of multiprogramming in the system. Even systems with a large amount of RAM (10s or 100s of gigabytes) often cannot simultaneously store the full address space of

every process in the system. As a result, an OS can make more efficient use of system resources by running processes with only parts of their virtual address spaces loaded in RAM.

Implementing Virtual Memory Using RAM, Disk, and Page Replacement

From the Memory Hierarchy Chapter, we know that memory references usually exhibit a very high degree of locality. In terms of paging, this means that processes tend to access pages of their memory space with a high degree of temporal or spatial locality. It also means that at any point in its execution, a process is not typically accessing large extents of its address space. In fact, processes typically never access large extents of their full address spaces. For example, processes typically do not use the full extent of their stack or heap memory space.

One way in which the OS can make efficient use of both RAM and CPU is to treat RAM as a cache for disk. In doing so, the OS allows processes to run in the system only having some of their virtual memory pages loaded into physical frames of RAM. Their other virtual memory pages remain on secondary storage devices, such as disk, and the OS only brings them into RAM when the process accesses addresses on these pages. This is another part of the OS's **virtual memory** abstraction — the OS implements a view of a single large physical "memory" that is implemented using RAM storage in combination with disk or other secondary storage devices. Programmers do not need to explicitly manage their program's memory, nor do they need to handle moving parts in and out of RAM as their program needs it.

By treating RAM as a cache for disk, the OS keeps in RAM only those pages from processes' virtual address spaces that are being accessed or have been accessed recently. As a result, processes tend to have the set of pages that they are accessing stored in fast RAM and the set of pages that they do not access frequently (or at all) stored on slower disk. This leads to more efficient use of RAM because the OS uses RAM to store pages that are actually being used by running processes, and doesn't waste RAM space to store pages that will not be accessed for a long time or ever. It also results in more efficient use of the CPU by allowing more processes to simultaneously share RAM space to store their active pages, which can result in an increase in the number of ready processes in the system, reducing times when the CPU is idle due to all the processes waiting for some event like disk I/O.

In virtual memory systems, however, processes sometimes try to access a page that is currently not stored in RAM (causing a **page fault**). When a page fault occurs, the OS needs to read the page from disk into RAM before the process can continue executing. The MMU reads a PTE's valid bit to determine whether it needs to trigger a page fault exception. When it encounters a PTE whose valid bit is zero, it traps to the OS, which takes the following steps:

1. The OS finds a free frame (e.g., frame j) of RAM into which it will load the faulted page.
2. It next issues a read to the disk to load the page from disk into frame j of RAM.
3. When the read from disk has completed, the OS updates the PTE entry, setting the frame number to j and the valid bit to 1 (this PTE for the faulted page now has a valid mapping to frame j).

4. Finally, the OS restarts the process at the instruction that caused the page fault. Now that the page table holds a valid mapping for the page that faulted, the process can access the virtual memory address that maps to an offset in physical frame j.

To handle a page fault, the OS needs to keep track of which RAM frames are free so that it can find a free frame of RAM into which the page read from disk can be stored. Operating systems often keep a list of free frames that are available for allocating on a page fault. If there are no available free RAM frames, then the OS picks a frame and replaces the page it stores with the faulted page. The PTE of the replaced page is updated, setting its valid bit to 0 (this page's PTE mapping is no longer valid). The replaced page is written back to disk if its in-RAM contents differ from its on-disk version; if the owning process wrote to the page while it was loaded in RAM, then the RAM version of the page needs to be written to disk before being replaced so that the modifications to the page of virtual memory are not lost. PTEs often include a **dirty bit** that is used to indicate if the in-RAM copy of the page has been modified (written to). During page replacement, if the dirty bit of the replaced page is set, then the page needs to be written to disk before being replaced with the faulted page. If the dirty bit is 0, then the on-disk copy of the replaced page matches the in-memory copy, and the page does not need to be written to disk when replaced.

Our discussion of virtual memory has primarily focused on the *mechanism* part of implementing paged virtual memory. However, there is an important *policy* part of paging in the OS's implementation. The OS needs to run a **page replacement policy** when free RAM is exhausted in the system. A page replacement policy picks a frame of RAM that is currently being used and replaces its contents with the faulted page; the current page is *evicted* from RAM to make room for storing the faulted page. The OS needs to implement a good page replacement policy for selecting which frame in RAM will be written back to disk to make room for the faulted page. For example, an OS might implement the **least recently used** (LRU) policy, which replaces the page stored in the frame of RAM that has been accessed least recently. LRU works well when there is a high degree of locality in memory accesses. There are many other policies that an OS may choose to implement. See an OS textbook for more information about page replacement policies.

Making Page Accesses Faster

Although paging has many benefits, it also results in a significant slowdown to every memory access. In a paged virtual memory system, every load and store to a virtual memory address requires *two* RAM accesses: the first reads the page table entry to get the frame number for virtual-to-physical address translation, and the second reads or writes the byte(s) at the physical RAM address. Thus, in a paged virtual memory system, every memory access is twice as slow as in a system that supports direct physical RAM addressing.

One way to reduce the additional overhead of paging is to cache page table mappings of virtual page numbers to physical frame numbers. When translating a virtual address, the MMU first checks for the

page number in the cache. If found, then the page's frame number mapping can be grabbed from the cache entry, avoiding one RAM access for reading the PTE.

A **translation look-aside buffer** (TLB) is a hardware cache that stores (page number, frame number) mappings. It is a small, fully associative cache that is optimized for fast lookups in hardware. When the MMU finds a mapping in the TLB (a TLB hit), a page table lookup is not needed, and only one RAM access is required to execute a load or store to a virtual memory address. When a mapping is not found in the TLB (a TLB miss), then an additional RAM access to the page's PTE is required to first construct the physical address of the load or store to RAM. The mapping associated with a TLB miss is added into the TLB. With good locality of memory references, the hit rate in the TLB is very high, resulting in fast memory accesses in paged virtual memory — most virtual memory accesses require only a single RAM access. Figure 11 shows how the TLB is used in virtual-to-physical address mappings.

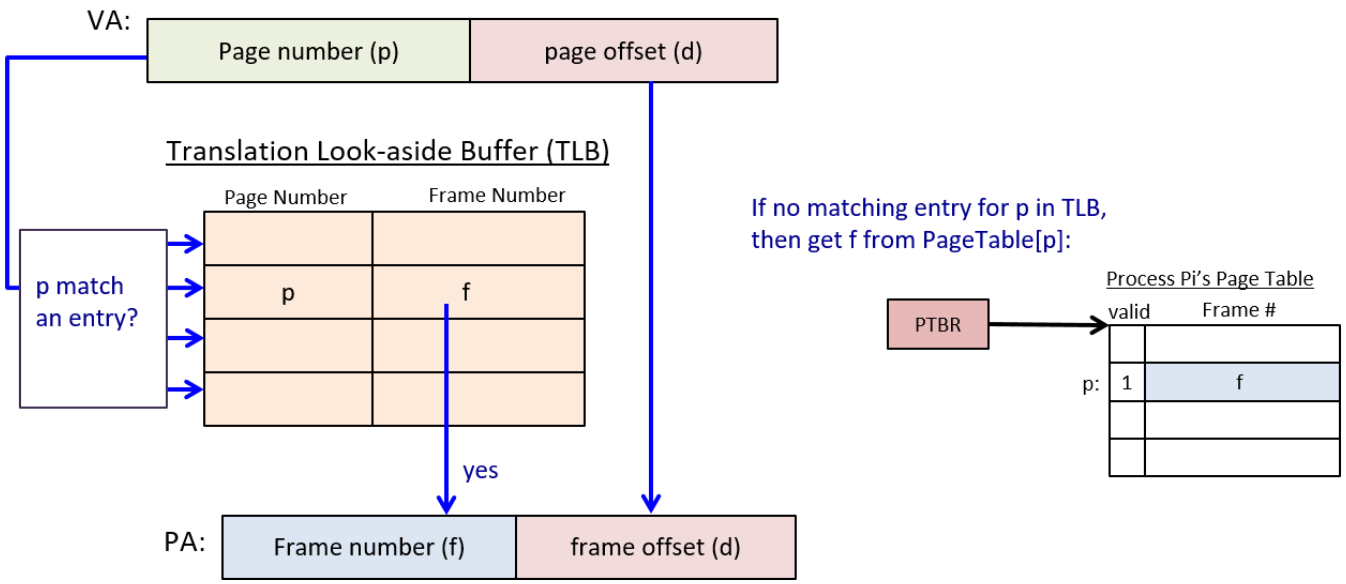

Figure 11. The translation look-aside buffer (TLB) is a small hardware cache of virtual page to physical frame mappings. The TLB is first searched for an entry for page p. If found, no page table lookup is needed to translate the virtual address to its physical address.

Contents