Introduction

We've assumed that address space is unreasonably small and fits into physical memory up to this point. We've been presuming that every running process's address space fits into memory. We'll now loosen these big assumptions and assume that we want to support multiple large address spaces running at the same time.
To accomplish so, we'll need to add a new memory hierarchy level. We've assumed that all pages are stored in physical memory so far. However, in order to handle big address spaces, the OS will want a location to store portions of address spaces that aren't currently in high demand. In general, such a place should have more capacity than memory; as a result, it is generally slower (if it were faster, we would just utilize it as memory, wouldn't we?). A hard disk drive usually fills this job in current systems. As a result, massive and slow hard drives are at the bottom of our memory structure, with memory slightly above. As a result, we've arrived at the essence of the issue:

How can the OS make use of a larger, slower hardware to create the illusion of a vast virtual address space while remaining transparent?

One question you could have is why we wish to support a process with a single huge address space. The answer is, once again, convenience and ease of usage. With a huge address space, you don't have to worry about whether your program's data structures will fit in memory; instead, you may create the program naturally, allocating memory as needed. It's a tremendous illusion provided by the OS, and it makes your life a lot easier. Thank you very much! Older systems with memory overlays, which required programmers to manually move portions of code or data in and out of memory as needed [D97], have contrast. Consider the following scenario: you must first arrange for the code or data to be in memory before invoking a function or accessing data; nasty!

Beyond a single process, swap space allows the OS to create the illusion of a vast virtual memory for numerous processes operating at the same time. The ability to swap out some pages was almost required by the invention of multiprogramming (running many programs "at once" to better utilize the system), as early machines simply could not keep all of the pages required by all processes at once. As a result, we wish to facilitate using more memory than is physically accessible due to the combination of multiprogramming and ease-of-use. It's something that all modern virtualization systems do, and it's something we'll learn more about in the future.

Swap Space

The first step is to save some disk space for paging. Operating systems call this swap space because it is used to swap pages out of memory and back in. This assumes the OS can read and write to swap space in page-sized units. To do so, the OS must remember a page's disk address.
The amount of the swap space defines the maximum number of memory pages that a system can use at any given time. Let us pretend that is pretty massive for now.

Figure 21.1 shows a 4-page physical memory and an 8-page swap space. In the example, three processes (Proc 0, 1, and 2) share physical memory, but only portion of their valid pages are in memory, with the rest in swap space on disk. So it's evident that the fourth process (Proc 3) isn't functioning. One swap block is left. The system can use swap space to make memory appear larger than it actually is.

Notably, swap space is not the only place on disk where data is swapped. Assume you are executing a binary program (e.g., ls, or your own compiled main program). This binary's code pages are stored on disk and loaded into memory when the application runs (either all at once when the program starts execution, or, as in modern systems, one page at a time when needed). If the system has to free up memory for other purposes, it can safely reuse the code pages' memory space, knowing it can subsequently swap them in from the on-disk binary in the file system.

The Present Bit

Now that we have disk space, we need to add some higher-level system machinery to handle page swapping. Assume, for simplicity, we have a hardware-managed TLB.

Begin by recalling memory references. Virtual memory references (for fetching instructions or data) are generated by the operating process and are translated into physical addresses by the hardware before fetching data from memory.
If the TLB finds a match (a TLB hit), it generates the physical address and gets it from memory. This is a common situation, as it is quick (requiring no additional memory accesses).

To search up a page table entry (PTE), the hardware locates the page table in memory (using the page table base register) and uses the VPN as an index. A TLB hit is generated if the page is valid and present in physical memory.

But to allow page swapping to disk, we need to install extra gear. This means that a page may not exist in physical memory when examined by hardware. In a software-managed TLB architecture, the OS decides this using a new piece of information in each page-table entry called the present bit. Assuming the present bit is set to one, the page is in physical memory and everything continues as before. A page fault occurs when a program accesses a page that isn't in physical memory.

The OS is called to handle a page fault. A page-fault handler runs and must service the page fault, as we will see.

The Page Fault

TLB misses can be handled by either hardware-managed TLBs or software-managed TLBs (where the OS does). In any case, if a page is missing, the OS is responsible for handling the page fault. The OS page-fault handler determines what to do. Even with a hardware-managed TLB, the hardware relies the OS to manage page faults.

A page fault will require the OS to swap in a page from disk if it is not present. So, how will the OS know where to look for the required page? In many systems, this data is stored in the page table. So the OS might use the PTE's data bits, like the page's PFN, to store a disk address. When the OS receives a page fault, it searches up the address in the PTE and requests the page from disk.

After the disk I/O, the OS will update the page table to reflect the new page, update the PFN field of the page-table entry (PTE) to reflect the new page's address in memory, and retry the instruction. In this case, the TLB miss would be serviced and the translation added (one could alternately update the TLB when servicing the page fault to avoid this step). Once the TLB translation is found, a final restart will fetch the appropriate data or instruction from memory at the translated physical location.

The process will be blocked while the I/O is running. So the OS can run other ready processes while the page fault is handled. Because I/O is expensive, overlapping I/O (page fault) of two processes is another method a multiprogrammed system might maximize its hardware.

What to do with Full Memory

In the example above, we assumed there was sufficient of free memory to page in a page from swap space. This isn't always the case; one's memory may be overburdened (or close to it). As a result, the OS may want to page out one or more pages before bringing in the new page(s). The page-replacement policy is the process of deciding which pages should be removed or replaced.

As it turns out, a lot of thinking has gone into developing a proper page-replacement policy, because removing the incorrect page can have a significant impact on program performance. Making the wrong option can cause a program to run at disk-like rates rather than memory-like speeds, which means a program could run 10,000 or 100,000 times slower in today's technology. As a result, such a policy is something we should investigate further; indeed, we will do so in the following chapter. For the time being, knowing that such a policy exists, based on the procedures explained here, is sufficient.

The handler then performs an I/O request to read the page from swap space with the physical frame in hand. Finally, the OS changes the page table and retries the instruction after the slow procedure is complete. A TLB miss will occur on the first retry, followed by a TLB hit on the second retry, at which point the hardware will be able to access the desired item.

When Replacements Happen

The method we've discussed replacements so far assumes that the OS waits until memory is completely full before replacing (evicting) a page to make room for another. As you might expect, this is unrealistic, and there are numerous reasons for the OS to maintain a tiny percentage of memory free more frequently.

Most operating systems feature some type of high watermark (HW) and low watermark (LW) to help decide when to start evicting pages from memory to preserve a small amount of memory free. When the OS detects that there are less LW pages available than there are LW pages available, a background process responsible for memory freeing runs. Until there are HW pages available, the thread evicts pages. The background thread, often known as the swap daemon or page daemon1, then sleeps, content that it has released some memory for running processes and the operating system to use.

New performance optimizations can be achieved by doing many replacements at the same time. Many systems, for example, will cluster or group a number of pages and write them out to the swap partition all at once, increasing disk efficiency; as we will see later when we discuss disks in greater detail, such clustering reduces seek and rotational overheads of a disk, resulting in noticeably improved performance.

The control flow in Figure 21.3 should be significantly adjusted to work with the background paging thread; instead of doing a straight replacement, the algorithm should simply check if there are any free pages available. If not, it will notify the background paging thread that free pages are required; once the thread has freed up some pages, it will re-awaken the original thread, which will then be able to page in the appropriate page and continue working.

Summary

We've covered the concept of accessing more memory than is physically available in a system in this short chapter. Because a present bit (of some kind) must be supplied to notify us whether the page is present in memory or not, greater complexity in page-table structures is required. If this isn't the case, the operating system's page-fault handler kicks in to handle the page fault, arranging for the transfer of the required page from disk to memory, maybe after replacing some pages in memory to create way for those that will be swapped in soon.

Remember, most critically, that all of these actions occur in full view of the process. In terms of the process, it's simply gaining access to its own private, contiguous virtual memory. Pages are stored at arbitrary (non-contiguous) positions in physical memory behind the scenes, and they are sometimes not even present in memory, necessitating a fetch from disk. While we anticipate that memory access is quick in most instances, it may require numerous disk operations in other cases; even something as basic as performing a single instruction can take many milliseconds in the worst scenario.