**Chapter 21: Beyond Physical Memory: Mechanisms**

We now wish to support many concurrently-running large address spaces. To do so, we require an additional level in the **memory hierarchy**.

Beyond just a single process, the addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently-running processes. The combination of multiprogramming and ease-of-use leads us to want to support using more memory than is physically available.

**21.1 Swap Space**

The first thing we will need to do is to reserve some space on the disk for moving pages back and forth. In operating systems, we generally refer to such space as **swap space.** Thus, we will simply assume that the OS can read from and write to the swap space, in page-sized units. To do so, the OS will need to remember the **disk address** of a given page.

The size of the swap space is important as ultimately it determines the maximum number of memory pages that can be in use by a system at a given time. We first assume that it is very large.

We should note that swap space is not the only on-disk location for swapping traffic. For example, if we are running a program binary, the code pages from this binary are initially found on disk, and when it runs, they are loaded into memory. if the system needs to make room in physical memory for other needs, it can safely re-use the memory space for these code pages, knowing that it can later swap them in again from the on-disk binary in the file system.

**21.2 The Present Bit**

If we wish to allow pages to be swapped to disk, however, we must add even more machinery. Specifically, when the hardware looks in the PTE, it may find that the page is not present in physical memory. To do so, the hardware check the **present bit** in PTE (1 for presenting in physical memory, 0 if not in memory but somewhere in the disk). The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.

A piece of code called **page-fault handler** will run to handle page fault

**21.3 The Page Fault**

When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.

When the disk I/O completes, the OS will then update the page table to mark the page as present, update the PFN field of the page-table entry (PTE) to record the in-memory location of the newly-fetched page, and retry the instruction.

**21.4 What If Memory Is Full?**

The OS might like to first **page out** one or more pages to make room for the new page(s) the OS is about to bring in. The process of picking a page to kick out or replace is known as the **page-replacement policy**.

**21.5 Page Fault Control Flow**

Text

Description automatically generated

When the page is both present and valid (18-21), the TLB miss handler can simply grab the PFN from the PTE, retry the instruction and continue. In the second case (Lines 22–23), the page fault handler must be run as it is not present in physical memory. Third (and finally), the access could be to an invalid page, due for example to a bug in the program (Lines 13–14). In this case, no other bits in the PTE really matter; the hardware traps this invalid access, and the OS trap handler runs, likely terminating the offending process.

Text, letter

Description automatically generated

From the software control flow, the OS roughly must do in order to service the page fault. First, the OS must find a physical frame for the soon-to-be-faulted-in page to reside within; if there is no such page, we’ll have to wait for the replacement algorithm to run and kick some pages out of memory, thus freeing them for use here.

**21.6 When Replacements Really Occur**

To keep a small amount of memory free, most operating systems thus have some kind of **high watermark** (HW) and **low watermark** (LW) to help decide when to start evicting pages from memory.

When the OS notices that there are fewer than LW pages available, a background thread that is responsible for freeing memory runs. The thread evicts pages until there are HW pages available. The background thread, sometimes called the **swap daemon** or **page daemon**, then goes to sleep, happy that it has freed some memory for running processes and the OS to use.