Virtual Memory: Mechanics of Page Faults

Hello, and Welcome to this lesson on the mechanics of virtual memory, where we explore in detail how on-demand paging is implemented.

We will start by reminding ourselves of what we learned about page-level swapping. As a side note: when we talk about "pages" we don't use the term "swapping" that much, it is used more for processes. We say "paging" instead.

We will then learn how this type of memory is represented in a real system with help of the page table.

Once we understand the role of the page table, we will be able to walk our way through what happens when the CPU tries to reference a memory location that is paged out. Now the structure of the page table entries start making sense. The "valid" bit, or "present" bit in the x86, is used to trigger a page-fault exception whenever the CPU tries to access a memory location that is not in memory, but rather is paged out to secondary storage.

We will learn that as part of a fault, the page has to be copied into a frame in physical memory before the CPU can access it. As long as there are empty frames available, this is no problem. If all frames are used, then the operating system has to pick a frame – we call this the "victim" – to remove from main memory in order to create space in main memory for the new frame. This step is called victim selection or frame selection.

We will see how the hardware can support the selection of victims such that I/O operations are minimized, with the use of the so-called "dirty" bit.

Often, it is best to select a victim that has not been used recently. Here again, the hardware often helps by providing a so-called "use" bit. Which in the x86 is called the "accessed" bit.

In future lessons we will call this process of selecting a victim and storing the new page in its location "page replacement", and many different page replacement algorithms are used across popular operating systems. When we will learn about these algorithms, we will refer back to the underlying mechanisms that we explore in this lesson.

Let's begin.

Recap: Page-Level on-Demand Swapping

Let's start by looking a system with a secondary device that is used to swap pages. Because we swap pages, we call it a "paging" device.

Let's have a process, we call it P1, with say 7 pages, Page 0, to Page 6. Pages 0, 2, 3, and 5 are stored in main memory.

And the remaining pages are stored on the paging device. We say that they are paged out.

The pages in the red rectangle are in the address space of the process.

In reality, of course, they are stored in frames in the physical memory. Here we have 12 frames, from Frame 0 to Frame 11.

In this case, Page 0 is stored in Frame 2, Page 2 is stored in Frame 8, Page 3 in Frame 3, and Page 5 in Frame 5.

Let's now add a second process, called P2, also with 7 pages, where pages 0, 1, 5, and 6 are stored in memory, and the remaining pages, that is pages 2, 3, and 4, are paged out.

Page Tables

This mapping, of course, is done by the page table. Here the one for Process P1.

The the page table entry 0, for example, contains frame number 2 to indicate that Page 0 is stored in Frame 2. Page table entry 2 contains frame number 8, and so on.

Process P2 of course has its own page table, which maps the pages of P2 to frames in memory.

Let's focus on Process P1. Actually, on its page table.

Memory Access without Page Fault

Let's see what happens when the CPU references a memory location.

In this case, the CPU references a memory location that is in Page 0.

Page 0 is stored in Frame 2, and the memory management unit successfully translates the logical address to the corresponding physical address in Frame 2. The memory access succeeds. No surprises here.

Page Fault Step-by-Step: "valid" Bit

Let's see what happens when the CPU access a memory location that is paged out.

In this case the CPU accesses a memory location in Page 6. This page is paged out.

How identify that this page is not in memory?

Let's use the x86 as an example. We learned earlier that each page table entry has, in addition to the frame number, a set of bits that contain meta-data about the page.

One of these bits, the "valid" bit, or "present" bit in x86 parlance, indicates whether the page table entry is valid.

So, each entry in the page table has a valid bit, which is set for all pages that are stored in memory. For the other pages, in particular those that are paged out, the bit is false.

If the CPU tries to access a memory location that is in a page with an invalid page table entry, a page fault exception occurs.

Step-by-Step Page Fault: Page Fault Exception

In our example, the page table entry for Page 6 is invalid.

A page fault exception, in the x86 this would be a exception 14, is raised, and the handler for this particular exception takes over in the operating system.

The page fault handler first checks whether this is a reference to a bona-fide memory location that happens to be paged out. If this is the case, the missing page is located on the paging device.

Once the page is found on the storage devices, it is loaded and copied into an available memory frame.

Now the page table entry for this page is updated to reflect that the page is now located in main memory.

In this example, the page was loaded into Frame 10, and we indicate that the page table entry is valid.

Once everything is updated, the page-fault exception handler returns, and the CPU re-issues the memory reference.

This time the page table entry is valid, and the memory location in physical memory is accessed successfully.

What if all Frames are Used? (Frame/Victim Selection)

Let's walk through the steps one more time, but this time, the main memory is full.

The CPU issues a memory reference to the page that is paged out, an exception occurs, the operating system takes over, and checks whether this is a bona-fide memory page. If so, the page is located on the paging device.

Unfortunately the main memory is full, and we need to free some space before we can copy the page from the paging device into main memory.

We need to select a victim, that is, we need to select a frame, whose content we page out in order to page in the needed page.

In this example, the victim is Frame 8, which contains Page 2 of the process.

We evict this page to the paging device.

Before we continue we need to update the page table to reflect the fact that Page 2 has been paged out.

For this we mark the page table entry for Page 2 to be invalid.

We now copy Page 6 into the newly freed frame, and continue from there. We call the steps of evicting a page in order to page in another page "page replacement", and the victim selection algorithm is an important aspect of the page replacement algorithm.

Minimizing I/O Writes with "Dirty" Bit

Let's have a closer look at the I/O operations needed for this last page fault.

First we have to evict the victim page to the paging device. This needs an IO WRITE operation, which is very expensive.

After this write we bring in the new page, which needs an IO READ operation, which is also very expensive.

There is not much we can do about the READ,

But we would like to avoid the WRITE operation whenever possible. How can this be done? One case where we clearly don't need to write a page to the paging device is if we did not modify it since the time when we paged it in last. If we left a copy on the paging device when we brought the page in, and we did not modify the page, then the two copies, meaning the one in memory and the other on the paging device, are identical. This would save us the WRITE operation.

Here the memory management hardware often comes in handy by maintaining meta-data on how memory pages have been accessed.

In the case of the x86 the page table entry maintains a so-called "dirty" bit, which is set by the hardware whenever the CPU writes to the page.

So, if our page table maintains such a dirty bit, and we clear this bit each time we page in a page, then we know whether we need to do a WRITE as part of the page replacement.

In the case of the selected victim, Page 2, the "dirty" bit is set, which means that we need to do a WRITE operation.

If the "dirty" bit were not set, we could save ourselves the WRITE .

Many page replacement algorithms therefore avoid selecting victim pages that have the dirty bit set.

Victim Selection with the "Use" Bit

So we said that one criteria for the victim selection in a page replacement algorithm is whether the page is dirty, which is represented by the dirty bit. Some algorithms prefer to select a page that has not been referenced recently.

Again, the hardware often has support for this. In the x86 the so-called "accessed" or "use" bit of the page table entry is set whenever a page is referenced by the CPU.

If our page table has access to such a bit, then we can keep track of which pages have been access since we last cleared the use bit. Recall that the page table is stored in memory. It is therefore very easy for the operating system to clear and check these bits in the page table.

For example, Page 2 has been referenced since we last cleared the use bit. It also is marked as dirty. As we will see in later lessons, this is a bad choice of page for replacement.

Page 3, on the other hand, has not been accessed since the use bit was cleared. Incidentally, the page is also clean. Depending on when the OS cleared the bit, it may decide that this is a good page to pick for replacement.

Virtual Memory: Mechanics of Page Faults

With this description of the "valid", "dirty" and "use" bit we are nearing the end of this lesson on mechanisms for page faults in virtual memory.

After reviewing the general concepts of on-demand paging, we looked at the role that is played by the memory management unit, in particular the page table.

We explored in detail what triggers a page fault, and what happens during a page fault.

We learned that page faults are typically triggered by the page table entry being marked as "invalid" using a reserved bit, the "valid" bit, in the entry.

In order to page in a page from the paging device we need to have space in memory to fit the new page. If the memory is full, an existing page in memory needs to be evicted before bringing in the new page. This is called "page replacement".

We learned how the hardware supports page replacement by maintaining meta-data that describes how the page has been accessed. For example, the "dirty" bit records whether a page has been modified and thus needs to be written to the paging device when replaced.

Finally, another example of hardware support is the "use" bit, which marks recently accessed pages and thus allows the operating system to pick those pages for replacement that have not been accessed recently.

We genuinely hope that you have enjoyed this lesson on mechanisms for virtual memory, and that you are ready to explore, in the following lessons, page replacement algorithms and overall page management, and that you will be able to reason about their relative benefits. Given the huge cost of a single page fault, page replacement algorithms that minimize the number of page faults are absolutely critical for the performance the system. This is therefore an exciting area to learn more about.

Thank you for watching!