【OS】Day26

Class	Operating System: Three Easy Pieces
Date	@January 25, 2022

【Ch21】Beyond Physical Memory: Mechanisms

Thus far, we've assumed that an address space is unrealistically small and fits into physical memory. We will now relax these big assumptions, and assume that we wish to support many concurrently-running large address spaces.

To do so, we require an additional level in the memory hierarchy. Thus far, we have assumed that all pages reside in physical memory.

However, to support large address spaces, the OS will need a place to stash away portions of address spaces that currently aren't in great demand. In modern systems, this role is usually served by a hard disk drive.

Thus, here is our crux: *How can the OS make use of a larger, slower device to transparently provide the illusion of a large virtual address space?*

The addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently-running processes.

The invention of multiprogramming almost demanded the ability to swap out some pages.

21.2 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 OS, we generally refer to such space as swap space, because we swap pages out of memory to it and swap pages into memory from it. 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.

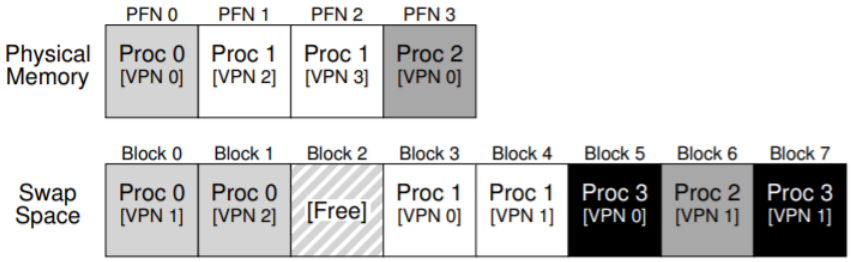

Figure 21.1: **Physical Memory and Swap Space**

From the figure above, we see a little example of a 4-page physical memory and an 8-page swap space.

In the example, three processes(Proc 0, Proc 1, and Proc 2) are actively sharing physical memory; each of the three, however, only have some of their valid pages in memory, with the rest located in swap space on disk.

A fourth process(Proc 3) has all of its pages swapped out to disk, and thus clearly isn't currently running.

21.2 The Present Bit

When the hardware looks in the PTE, it may find that the page is not present in physical memory. The way the hardware determines this is through a new piece of information in each page-table entry, known as the present bit.

- If the present bit is set to one, it means the page is present in physical memory and everything proceeds as above
- If it is set to zero, the page is not in memory but rather on disk somewhere.

The act of accessing a page that is not in physical memory is commonly referred to as a page fault.

Upon a page fault, the OS is invoked to service the page fault. A particular piece of code, known as a page-fault handler, runs, and must service the page fault, as we now describe.

21.3 The Page Fault

If a page is not present, the OS page-fault handler runs to determine what to do.

How will the OS know where to find the desired page?

In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address.

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.

This next attempt may generate a TLB miss, which would then be serviced and update the TLB with the translation.

Note that while the I/O is in flight, the process will be in the blocked state. Thus, the OS will be free to run other ready processes while the page fault is being serviced.

21.4 What if Memory is Full?

In the process described above, we assumed there is plenty of free memory in which to page in a page from swap space.

Of course, this may not be the case; memory may be full.

Thus, the OS might like to first page out one or more pages to make room for the new pages 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.

Making the wrong decision can cause a program to run at disk-like speeds instead of memory-like speeds.

21.5 Page Fault Control Flow

The Complete Flow of Memory Access

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else
16          if (CanAccess(PTE.ProtectBits) == False)
17              RaiseException(PROTECTION_FAULT)
18          else if (PTE.Present == True)
19              // assuming hardware-managed TLB
20              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21              RetryInstruction()
22          else if (PTE.Present == False)
23              RaiseException(PAGE_FAULT)
```

Figure 21.2: **Page-Fault Control Flow Algorithm (Hardware)**

There are now three important cases to understand when a TLB miss occurs:

1. First,(Lines 18-21) that the page was both present and valid; in this case, the TLB miss handler can simply grab the PFN from the PTE, retry the instruction, and thus continue
2. In the second case(Lines 22-23), the page fault handler must be run as the asked page in not currently residing in memory.

3. Third, the access could be to an invalid page, due, for example, to a bug in the program(Lines 13-14).

```
PFN = FindFreePhysicalPage()
if (PFN == -1)              // no free page found
    PFN = EvictPage()       // run replacement algorithm
DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
PTE.present = True          // update page table with present
PTE.PFN     = PFN           // bit and translation (PFN)
RetryInstruction()          // retry instruction
```

Figure 21.3: **Page-Fault Control Flow Algorithm (Software)**

The figure above shows the control flow of the page-fault handler.

1. First, the OS must find a physical frame for the soon-to-be-faulted-in page to reside within;.
2. If there is no such page; we'll have to wait for the replacement algorithm to run and kick some pages out of memory.