**VIRTUAL MEMORY**

Use main memory as a “cache” for secondary (disk) storage

* Managed jointly by CPU hardware and the operating system (OS)

Programs share main memory

* Each gets a private virtual address space holding its frequently used code and data
* Protected from other programs

CPU and OS translate virtual addresses to physical addresses

* VM “block” is called a page
* VM translation “miss” is called a page fault

Memory’de data bulunamadığında ve disc access yapılacağında bekletilecek olan program CPU’dan çekilir ve schedule’ı OS alır. OS, data memory’de en iyi nereye yerleştirilebilir buna karar veriyor. Çünkü zaten çok bekliyoruz, OS bu sırada data main memory’de nereye yerleştirilecek, kim silinecek, silinen aşağıya nasıl yazılacak hepsini handle edebiliyor.

Problems: Bütün programlar aynı main memory’yi kullanıyor. A programı main memory’yi kullanırken, B programı A programının kullandığı memory lokasyonlarına girdiği anda bütün data güvenliği ortadan kalkar. Onun için main memory’de programın nerede olacağının programcı tarafından bilinmemesi lazım.

Mesela A processi 5000-8000 arası adresleri kullanmak istiyor, B processi de 7000-9000 arası adresleri kullanmak istiyor. Aynı adresleri kullanan iki processi aynı anda çalıştıramazsın. Ancak “beni çalıştırmak için şunu kapatman lazım” gibi bir uyarıyı hiç görmeyiz. Dolayısıyla main memory’yi bütün processler arasında paylaştırmamız lazım.

Program hiçbir zaman main memory adresleriyle çalışmaz.

Bunu virtual adreslerle çözeceğiz.

Virtual memory size 4 GB diyelim. Bütün processlere “senin 4 GB yerin var, başla 0’dan 232-1’e (4 GB) kadar olan adresler senin.” ya da “sana 1 GB yer veriyorum, 0’dan 230-1’e kadar” diyor. Normalde bu sistemin patlaması lazım çünkü o kadar yerimiz yok.

Aslında çoğu program kendisine verilen 1 GB’ı kullanmıyor. Biri 256 B tutuyor, diğeri 5 KB tutuyor. Eğer program gerçekten 1 GB kullanıyorsa o zaman 4 program açtığında sistem kitlenir.

Gerçek memory adreslerini program bilmediği için, gerçek memory adreslerini başka programlardan OS koruyabiliyor.

Linuxta yaptığın chmod ile access writeları değiştirmek demek virtual memory sistemine o program çalışırken oraya kimlerin access edebileceğini söylemek demek.

Main memory’de artık blocklardan değil, birçok blocktan oluşan pagelerden söz ederiz. Çünkü block da küçük kalıyor çünkü diske gitmek çok zor. Gittiysem daha büyük bir şey alayım.

Virtual memory’de miss varsa (aradığın page’i main memory’de bulamadın), buna page fault denir.

What is virtual memory?

Diagram

Description automatically generated



Programın hepsi bir anda main memory’de olmak zorunda değil.

A programının kastettiği 500 numaralı word ile B programının kastettiği 500 numaralı wordün main memoryde karşılığı farklı. Virtual adresten physical adrese dönüşüm yapman lazım. Buna address translation denir, bunu yapan tabloya da page table denir.



Page table: virtual adresleri ya physical adrese çevirir, ya da diskteki yerini gösterir.

Virtual adresler: programın kullanmak istediği tüm address space. Page table: Programa 1 GB adres verdiysen o 1 GB’lık virtual adreslerin her bir adresi için (program artık kaçını kullanıyorsa) ya bunun karşılığı physical memory’de şu adres diyor ya da bunun karşılığı diskte şu adres diyor.

Virtual memory (VM) allows main memory (DRAM) to act like a cache for secondary storage (magnetic disk).

VM address translation provides a mapping from the virtual address of the processor to the physical address in main memory and secondary storage.

VM provides the following benefits

* Allows multiple programs to share the same physical memory
* Allows programmers to write code (or compilers to generate code) as though they have a very large amount of main memory
* Automatically handles bringing in data from disk

Cache terms vs. VM terms

* Cache block => page
* Cache miss => page fault

block offset – index – tag diye ayırdığımız aslında physical adresti. Physical adreslerle çalıştığımızı varsaydık.

Main memory’den silinecek şeyler en uzun süredir orada olan şeylerdir. Yazılım kullandığın için bunu daha zekice anlayabilirsin.

Virtual and Physical Addresses

A virtual address consists of a virtual page number and a page offset.

The virtual page number gets translated to a physical page number.

The page offset is not changed

Diagram

Description automatically generated



Virtual address:

* Bir kısmı page’in içerisindeki hangi wordü, byte’ı istediğimizi söyleyen kısım.
* 4 KB page varsa virtual adresin least sig 12 biti hangi byte’ı istediğimizi söyler

Virtual adres 32 bitse, virtual memory size 232 byte yani 4 GB’tır. Herkese 4 GB söz veriyoruz.

Virtual page number büyüdükçe performans azalır.

Page offset physical adrese olduğu gibi yapıştırılır.

Virtual adres 32 bit iken physical adres 30 bitten oluşuyor. Yani main memory aslında 1 GB.

Bir tane program bile bana söz vermiştin diyip 4 GB’ın 1 GB’ını kullanmaya kalkarsa o zaman patlarız.

Physical adres ile cache’te veya main memory’de aradığımız datayı buluruz.

Address Translation

Diagram

Description automatically generated

Page Tables

Stores placement information

* Array of page table entries, indexed by virtual page number
  + 20 bitlik virtual page numberın varsa page tabledaki entry sayısı 220
  + Her virtual page number direkt sana page tableda bakacağın yeri gösteriyor, tag falan yok
  + Her farklı virtual page number için page table’da bir entry vardır
* Page table register in CPU points to page table in physical memory

If page is present in memory

* PTE (page table entry) stores the physical page number
* Plus other status bits (referenced, dirty, ...)
  + Örneğin değişiklik yapıldığında main memory ne kullanacak? Write back kullanması mantıklı olur çünkü diske ulaşmak çok maliyetli. Değişikliklerin hepsi main memory’de yapılır.
  + Main memory’de değişiklik yaptığın page atılacağı zaman, page dirty ise yani üzerinde değişiklikler yapılmışsa diske yazarız.

If page is not present

* PTE can refer to location in swap space on disk

Page table nerede tutuluyor?

* 20 bitlik virtual page numberın varsa page tabledaki entry sayısı 220 dedik
* Entryler içerisinde physical page number taşıyor, access write ve yardımcı bitler de taşıyor
* Bunlarla beraber bir page table entrysinin 32 bit (4 B) olduğunu düşün
* 220 x 4 byte = 4 MB 🡪 page table
* Cacheler KB mertebesinde yani page table’ı cachelerde tutamayız.
* Saklayabileceğimiz en hızlı yer: main memory
* CPU’nun aradığı datayı cache’te bulması için page table’a ihtiyacı var ve bu table main memoryde.
* Her accesste main memory’ye gideceğim, physical adres isteyeceğim.
* Bu büyük bir yavaşlama demek, düzeltilmesi lazım.

Address Translation with Page Tables

A page table translates a virtual page number into a physical page number.

The virtual page number is used as an index into the page table that contains

* The physical page number
* A valid bit that indicates if the page is present in main memory
  + Valid bit 0’sa page fault olur.
* A dirty bit to indicate if the page has been written
* Protection information about the page (read only, read/write, etc.)

Since page tables contain a mapping for every virtual page, no tags are required.

Page Table Diagram

Diagram

Description automatically generated

Page table register: Page table’ın main memory’deki başlangıç adresini tutar. Processor hangi programı çalıştırıyorsa, o programın page table başlangıç adresini bu registerda tutuluyor.

Page table’da bakmamız gereken yerin adresini bulmak için yapmamız gereken şey virtual page number ile page table register içerisindeki değeri toplamak ve o yere bakmak.

Geçmişte adres derken biz hep physical adresten bahsettik.

Accessing Main Memory or Disk

If the valid bit of the page table is zero, this means that the page is not in main memory.

In this case, a page fault occurs, and the missing page is read in from disk.

Diagram

Description automatically generated

Valid biti 1’se main memory’deki yerleri tutulur.  
Valid biti 0’sa diskteki yerleri tutulur.

Determining Page Table Size

Assume

* 32-bit virtual address 🡪 232 = herkese 4 GB söz veriyoruz
* 30-bit physical address 🡪 230  = 1 GB main memory
* 4 KB pages => 12 bit page offset
* Each page table entry is one word (4 bytes)

How large is the page table?

* Virtual page number = 32 - 12 = 20 bits
* Number of entries = number of pages = 220
* Total size = number of entries x bytes/entry = 220 x 4 = 4 Mbytes

*Each process running needs its own page table*

Since page tables are very large, they are almost always stored in main memory, which makes them slow.

Caching Virtual Addresses

Virtual memory seems to be really slow:

* Must access memory on load/store -- even cache hits!
* Worse, if translation is not completely in memory, may need to go to disk before hitting in cache!

Solution: Caching!

* Keep track of most common translations and place them in a “Translation Lookaside Buffer” (TLB)
* Bu ayrı cachein tek görevi içinde page table entrylerini tutmak.
* Bir page’e access ettiysen genelde uzun süre boyunca aynı page üzerinde çalışırsın çünkü page zaten büyük bir veri.
* Dolayısıyla TLB’ın çok fazla entry tutmasına gerek yok. Son access edilen örneğin 16 page tutsa bile süper.

Diagram, schematic

Description automatically generated

Tüm işlemler için Physical Address’e (PA) ihtiyaç var.

PA’e ulaşmak için translationa ihtiyaç var.

Translation için main memory’ye ihtiyaç var.

Making Address Translation Practical: TLB

Virtual memory => memory acts like a cache for the disk

Page table maps virtual page numbers to physical frames

Translation Look-aside Buffer (TLB) is a cache for translations

Diagram

Description automatically generated

Diyelim page tabledaki fiziksel karşılığı 2 olan page’e access etmek istedik. Sonra 5’e vs.

Bu accessleri TLB alıyor. Yani page’e ilk accessimizde tabii ki memory’ye gidiyoruz. Ancak ilk accessimizde sadece page table’dan entry’yi okumuyor, aynı zamanda bu entry’yi TLB’ye taşıyoruz. Yine bu entry’ye access etmek istediğimizde TLB’ye de bakıyoruz, karşılık gelen physical page number’ı alıyoruz ve bu physical page number ile page offseti birleştirip physical addresse ulaşabiliyorsun.

Translation-Lookaside Buffer (TLB)

A TLB acts as a cache for the page table, by storing physical addresses of pages that have been recently accessed.

Diagram

Description automatically generated

TLB’de tag de tutuyoruz çünkü artık virtual page number bir index göstermiyor. TLB’de herhangi bir yere yerleştirilmiş olabilir. Çünkü TLB full associative.

Virtual page number, TLB’nin tagi olarak saklanır.

Valid bit 1’se ve virtual page number tag ile eşitse, TLB’den aldığın physical page number direkt physical address oluşturmak için kullanılabilir.

Page fault varsa TLB’de zaten yoktur.

MIPS R2000 TLB and Cache

Diagram, schematic

Description automatically generated

CPU, aradığı adresi TLB’de bulamazsa veya aradığı datayı cache’te bulamazsa main memory ile muhattap oluyor.

TLB and Cache Operation

On a memory access, the following operations occur.

Diagram

Description automatically generated

TLB Misses

If page is in memory

* Load the PTE from memory and retry
* Could be handled in hardware
  + Can get complex for more complicated page table structures
* Or in software
  + Raise a special exception, with optimized handler

If page is not in memory (page fault)

* OS handles fetching the page and updating the pagetable
  + OS page table’a bakarak page’i diskten bulur.
  + Page’i main memory’ye taşıyor ve en iyi yere yerleştiriyor.
* Then restart the faulting instruction

TLB Miss Handler

TLB miss indicates

* Page present, but PTE not in TLB
* Page not present

Must recognize TLB miss before destination register overwritten

* Raise exception

Handler copies PTE from memory to TLB

* Then restarts instruction
* If page not present, page fault will occur

Handling TLB Misses and Page Faults

When a TLB miss occurs either

* Page is present in memory and update the TLB
  + occurs if valid bit of page table is set
* Page is not present in memory and O.S. gets control to handle a page fault

If a page fault occur, the operating system

* Access the page table to determine the physical location of the page on disk
* Chooses a physical page to replace - if the replaced page is dirty it is written to disk
* Reads a page from disk into the chosen physical page in main memory.

Since the disk access takes so long, another process is typically allowed to run during a page fault.

Page Fault Handler

Use faulting virtual address to find PTE

Locate page on disk

Choose page to replace

* If dirty, write to disk first

Read page into memory and update page table

Make process runnable again

* Restart from faulting instruction

Cache and Main Memory Parameters (Old)

Table

Description automatically generated

4 Qs for Virtual Memory

Q1: Where can a block be placed in the upper level?

* Miss penalty for virtual memory is very high
  + So full associativity is used
* Have software determine location of block while accessing disk
* Allow blocks to be place anywhere in memory (fully associative) to reduce miss rate.

Q2: How is a block found if it is in the upper level?

* Address divided into page number and page offset
* Page table and translation buffer used for address translation

Q3: Which block should be replaced on a miss?

* Want to reduce miss rate & can handle in software
* Least Recently Used typically used

Q4: What happens on a write?

* Writing to disk is very expensive
* Use a write-back strategy

TLB organization: include protection

TLB:

Table

Description automatically generated with low confidence

TLB usually organized as fully-associative cache

* Lookup is by Virtual Address
* Returns Physical Address + other info

Dirty => Page modified (Y/N)?

Ref => Page touched (Y/N)?

Valid => TLB entry valid (Y/N)?

Access => Read? Write?

ASID => Which User?

Memory Protection

With multiprogramming, a computer is shared by several programs or processes running concurrently

* Need to provide protection
* Need to allow sharing

Mechanisms for providing protection

* Provide both user and supervisor (operating system) modes
  + supervisorsan access writeların hepsi için açık olur
* Provide CPU state that the user can read, but cannot write
  + user/supervisor bit, page table pointer, and TLB
* Provide method to go from user to supervisor mode and vice versa
  + system call or exception : user to supervisor
  + system or exception return : supervisor to user
* Provide permissions for each page in memory
* Store page tables in the operating systems address space - can’t be accessed directly by user.

Virtual Memory Summary

Virtual memory (VM) allows main memory (DRAM) to act like a cache for secondary storage (magnetic disk).

Page tables and TLBs are used to translate the virtual address to a physical address

The large miss penalty of virtual memory leads to different strategies from cache

* Fully associative
* LRU or LRU approximation
* Write-back
* Done by software