Speeding up paging….

Vi har sett ved forelesning “Minne II” om at vi har memory lookup, hvor hver prosess blir utdelt enhver sin egen virtuelle minne adresse(egen page table). Det kan forekomme tilfeller hvor noen prosesser har utrolig mange adresser i tabellen som kan føre til at oppslag kan ta lang tid med tanke på hvor mange adresser som må gå igjennom i en page-table.

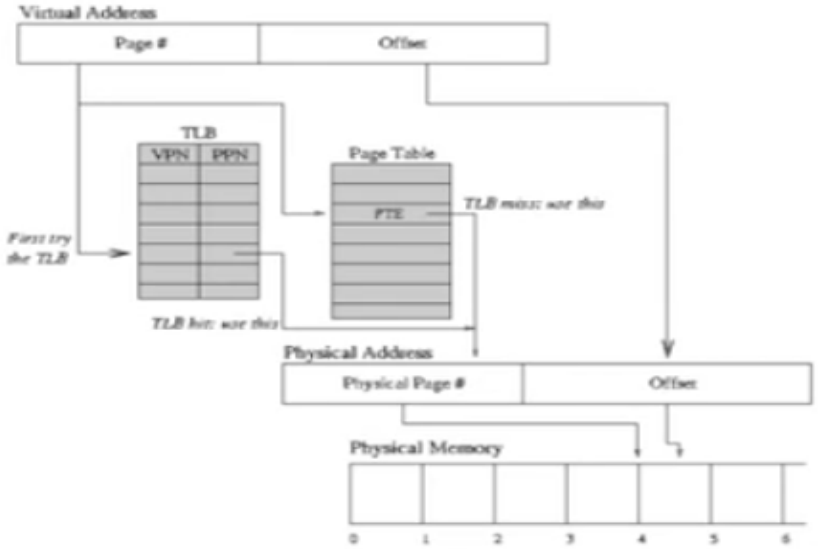
Large tables:

- 32-bit addresses, 4 KB pages → 1.048.576 entries
- 64-bit addresses, 4 KB pages → 4.503.599.627.370.496 entries

Ser vi bildet over, så kan det lang tid å mappe fra et virituelt minneadresse til det fysiske hvis vi har 64-bit adresser og at hver page har 4 kb. Derfor er en løsning på dette “Translation lookaside buffers” (aka associative memory).

Translation lookaside buffers(aka associative memory)

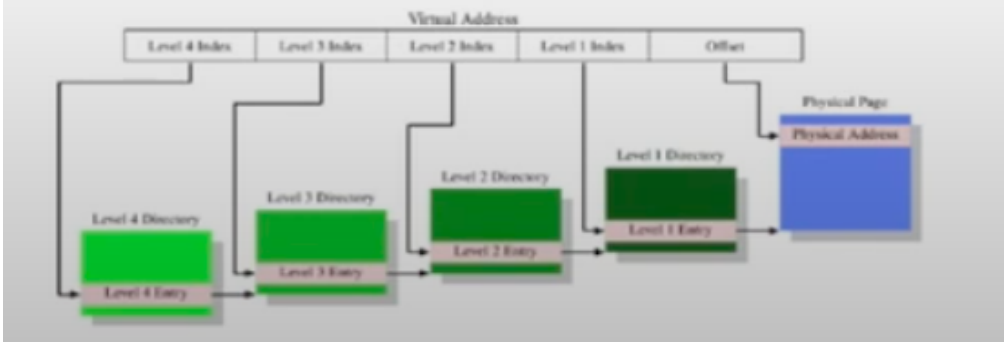
Det fungerer ved at en page-tabell vil ha “Hardware Caches” hvor disse “Cachene” vil oppbevare de nylig brukte pagesene. Da kan vi benytte av “Cache” til å aksessere nylig brukte pages for å redusere tiden det tar å måtte slå opp noe, når vi bare kan aksessere “Cache”. Dette gjør oppslag raskere, men avhengig av at pagen faktisk ligger i “Cache”.



Page size

En annen mulighet er å redusere nummeret av pages i slike store tabeller for å redusere tiden for oppslag.

Multi-level page tables



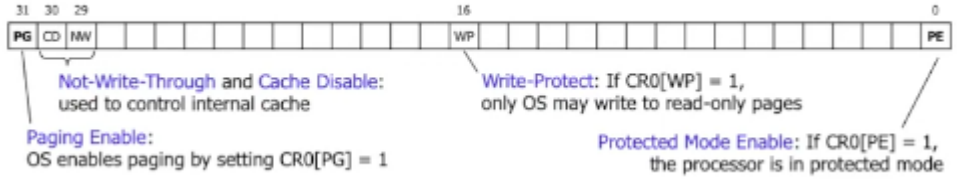
Et annet alternativ er å dele opp page-tabellen i "levels" hvor disse "levelene" peker på nye "page-table" for hvor adressene kan befinne seg. Grunnen til at vi har slike tables, er å forestille seg følgende: La oss si at vi har en 64-bit arkitektur, da må en page-tabell oppbevare mange milliarder av pages i en gitt tabell. Det betyr at oppslag mellom pages vil ta utrolig lang tid siden vi kun har en page-tabell som vi så i bildet helt på starten. Derfor kan vi benytte av "Multi-level-page-tables" til å hjelpe oss med å dele opp mulige adresser i ulike "levels". Ganske selvforklarende som man ser i bildet også, men om vi ønsker å hente en spesifikk "page" så vil man basere "page-number" for hvilken "level" som "pagen" befinner seg i, for å hente "pagen" vi er ute etter.

Paging on Pentium

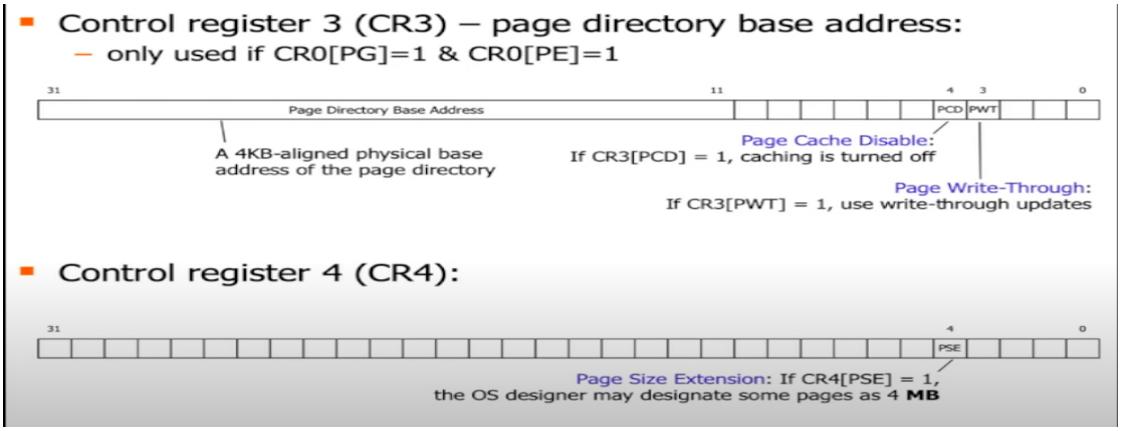
- The executing process has a 4 GB address space (2^{32}) – viewed as 1M (2^{20}) 4 KB (2^{12}) pages
  - The 4 GB address space is divided into 1 K page groups (pointed to by the 1 level table – page directory)
  - Each page group has 1 K 4 KB pages (pointed to by the 2 level tables – page tables)
- Mass storage space is also divided into 4 KB blocks of information
- Control registers: used to change/control general behavior (e.g., interrupt control, switching the addressing mode, **paging control**, etc.)

Dette er eksempelet som ble brukt som gav innledning for forståelsesoppgaven som involverte kontrollregisterne. Kontrollregistrene kontrollerer oppførselen ved pages, f.eks om det er tillatt å utføre en page, endre adresse-modusen og diverse.

- Control register 0 (CR0):

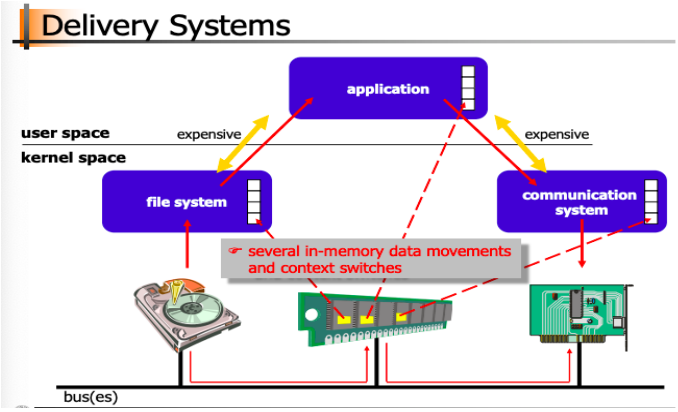


- "Control register 0 (CR0)" kontrollerer om paging er tillatt eller ikke. Det er to bit som må være satt for at paging er tillatt. Disse er "paging-enable [PG]" og "protected mode enable [PE]" for at pages kan benyttes.
- "Control register 1 (CR1)" eksisterer ikke lenger så denne kan sees vekk fra
- "Control register 2 (CR2)" avhenger av om [PG] og [PE] er satt til "1". Hvis begge er satt så vil denne registreren returnere den 32-bits adressen som gav oss en page-fault eller den fysiske adressen for den pagen.

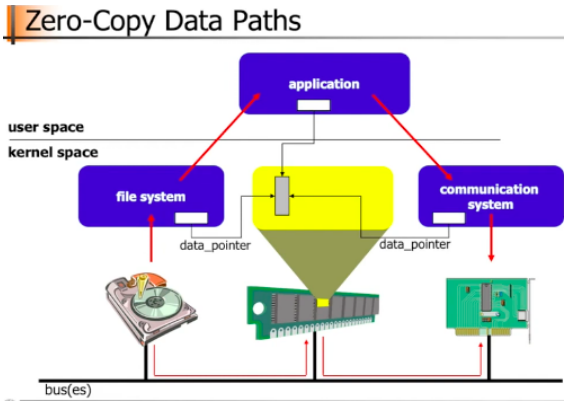


- "Control register 3 (CR3)" inneholder startadressen til page-directory. Dette er da første-nivået av page-tabeller som inneholder pekere av nye pages. Vi trenger dermed de øverste "20" bitene, for å peke på første byten av den tabellen.
- "Control register 4 (CR4)" har en bit kalt "PSE" som beskriver størrelsen for en page.

In-Memory Copy Operations

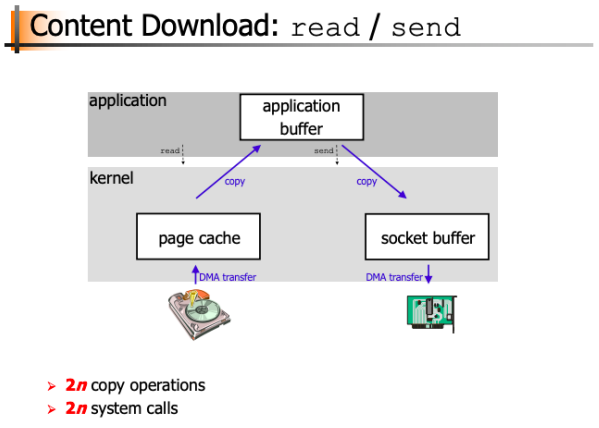


Bildet over, viser at de ulike komponentene over, vil håndtere de ulike delene som er av "kernel-space" eller "user-space". Her ser vi at fil-systeme vil aksessere minnet eller i verste fall disken som vil overføre en kopi av dataene over til applikasjonen. Applikasjonen vil da ta imot dataene og videreføre en kopi av disse dataene til kommunikasjonssystemet siden den ikke kan direkte overføre data til nettverkskortet på grunn av privilegienivåer. Men slike overføringer som vist over, er ganske dyre. En mulig løsning for dette er som vist under.



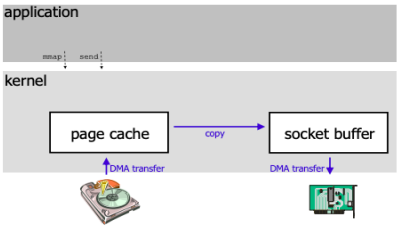
Det ble gjort en forskingsresultat i 90-tallet, hvor vi da har en fysisk enhet som alle kan aksessere dataene gjennom pekere som vi ser. Dette skal bli nyttig senere.

Laste ned "Content"



Dette kan ta tid for å måtte overføre kopier til et applikasjonsbuffer hvor bufferet vil da videresende denne kopien over til nettverkskortet. Vi så tidligere at dette kan være kostbart og dyrt med tanke på antall kopioperasjoner og system kall i bildet.

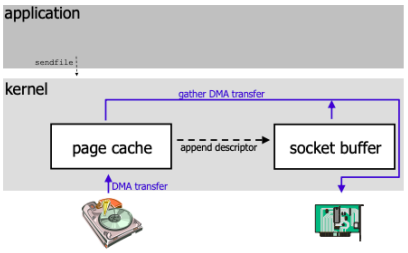
Content Download: `mmap` / `send`



- ***n*** copy operations
- **1 + *n*** system calls

Her ser vi at dette er bedre siden vi kan da bare sende en kopi direkte over fra disk til nettverkskortet gjennom mapping og send. Men vi har et bedre alternativ enn dette:

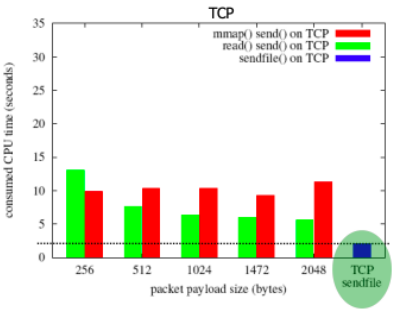
Content Download: `sendfile`



- ***0*** copy operations
- **1** system calls

Ved å bruke “zero-copy” så vil vi ha ingen kopioperasjoner og kun et system kall siden vi benytter av pekere som kan da aksessere dataen på den måten.

- Tested transfer of 1 GB file on Linux 2.6



Her ser vi resultatene ved bruk av de ulike operasjonen over hvor “sendfile” altså den “zero-copy-data” var best av dem alle.