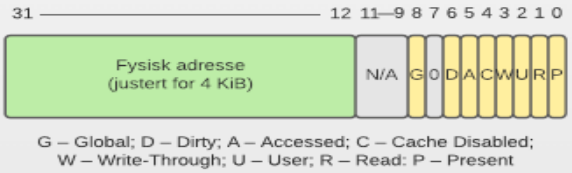
Multilevel Paging

Som teknikk løser virtuelt minne visse problemer, men introduserer også noen nye. Et abstraksjonslag legges over fysisk minne, slik at applikasjoner (eller snarere de som skriver dem) ikke behøver å bry seg med de rent fysiske detaljene ved minnehåndtering; operativsystemet gjør den tunge jobben i så måte. Sidetabellen, som knytter virtuelle adresser til fysiske adresser, brukes for å utføre denne funksjonen. Den bærer likevel med seg en viss kostnad i form av tid og rom: Sidetabellen må nødvendigvis lagres noensteds, mens sidetabelloppslag forbruker CPU-sykler.



G – Global; D – Dirty; A – Accessed; C – Cache Disabled;
W – Write-Through; U – User; R – Read: P – Present

Figur 1: Sidetabelloppføring

En sidetabelloppføring på IA-32-arkitekturen er vist i Figur 1. Gitt at sidestørrelsen er 4 KiB, hvor stor er sidetabellen? (Husk at antall sider kan utledes fra sidestørrelsen.)

Med en sidestørrelse på 4 KiB er hver side 4096 = 2^12 bytes. Da har man 20 bits til overs i adressen (32-12=20), som gir totalt 2^20=1 048 576 sider. Totalt får sidetabellen en størrelse på 1048576 * 4 = 4 194 304 bytes = 4 MiB.

Hvis det viser seg at sidetabellen ikke er overdrevent stor, hvorfor ikke lagre alt dets innhold i minnet til enhver tid? Med andre ord: Hvorfor bry seg med *multilevel paging* i utgangspunktet? Hvis det behøves, kan ikke sidetabellen simpelthen swappes ut til disk?

Hvis Carsten tar utgangspunkt i at størrelsen på sidetabellen til en prosess er på noen MB, så er ikke dette nødvendigvis så mye på så moderne maskin som Carstens Mac. Men hvis Carsten begynner å ha mange prosesser av gangen med hver sin sidetabell, for eksempel

100 stk, vil dette medføre et behov for noen hundre MB, som er mye unødvendig lagringsplass å bruke i det fysiske minnet.

Multilevel paging forhindrer dette; i stedet for at en hel sidetabell må være lastet i minnet, gjør man slik at noen bits i adressen peker til en mindre top-level tabell, som så holder minnelokasjoner til de lavere tabellene i et tabellhierarki. Totalt får man like mange sider i tabellhierarkiet som man kan bruke, men for tabellene på lavere nivå, trenger man ikke til enhver tid å ha alle lastet inn i minnet; kun den man søker etter ved en gitt minneoperasjon.

Man kan swappe deler av sidetabellen ut til disk; de sidetabellene på lavere nivå som ikke refereres til av en virtuell adresse, kan ligge på disken inntil man vil hente noe ut av en av dem. Dette er hva multilevel paging går ut på. Hvis man derimot skal swappe ut og inn en hel sidetabell, vil dette medføre enorme tids- og prosesseringskostnader, som er særdeles lite gunstig siden det, med multilevel paging, ikke koster mye minneplass å i hvert fall beholde hovedtabellen for en gitt prosess i minnet, selv når den ikke er aktiv.

Forandrer multilevel paging situasjonen når det gjelder å swappe ut sidetabellen? Hvordan i så fall?

Med Multilevel paging, kan en sidetabell peke på ene eller flere andre sidetabeller(eller de faktiske fysiske minnesidene). På denne måten får vi raskere oppslag ettersom tabellene er mindre. Hvis vi skal slå opp minneområder i et sidetabell, så vil vi da slå opp i mer enn en sidetabell for å finne minneområdet. I motsetning til før hvor vi kun har en tabell. De av tabellene som blir lite brukt kan lagres på disk om nødvendig, slik at swapping brukes ganske effektivt.

CPU-ens kontrollregistere, CR0, CR2 og CR3 på IA-32-arkitekturen, spiller viktige roller for sidetabelloppslag. Gi en sammenfatning av oppslagprosessen med fokus på hvordan disse registrene brukes.

I Control register 0 må Paging Enable (PG) og Protected Mode Enable (PE) være satt for at systemet skal bruke paging. Control register 2 inneholder den adressen som ga en page fault, avhengig av at PG og PE er satt i CR0. Control register 3 inneholder startadressen til page directory, altså det første nivået av page-tabeller. CR3 avhenger også av at PG og PE er satt i CR0.

Gruppen manglet å begrunne sammenfatning av oppslagsprossen ved tilfelle hvor vi har en page som peker på en adresse som ikke ligger i minne, men i disk. Gruppen under hadde en antakelse som så vekk fra page-fault og ble ansett som riktig.

(Jeg antar at det her blir spurt om et vanlig oppslag og ikke en page fault). Når vi gjør et oppslag av en virtuell adresse (ved bruk av multilevel paging), må vi slå opp i potensielt flere tabeller for å få den fysiske adressen. Kontrollregistrene CR0, CR2 og CR3 vil da inneholde data om oppslaget. Registret CR0 har bl.a. bitene, PG og PE som begge må være satt for at paging skal være skrudd på. Hvis dette er tilfældet vil CR2 inneholde den 32 bit strengen vi ønsker å slå opp. Vi ser da på de øverste 10 bitene i den virtuelle adressen for å finne elementet til den første tabellen. CR3 vil da inneholde page directory adressen, altså sidetabellens base-adresse, og vi bruker de 10 øverste bitene fra CR2 til å finne adressen til den siden der neste sidetabell er lokalisert. Vi antar at present bittet på denne siden også er satt og vi bruker de neste 10 bitene som en index i denne andre sidetabellen. Her antar vi også at present biten er satt og vi sitter nå igjen med den fysiske adressen til den siden vi ønsker å aksessere. De 12 bakerste bittene i CR2 blir da brukt som offset for å finne den nøyaktige adressen i det fysiske minnet.

Disker

Disker tilbyr billig lagring med høy kapasitet. De har eksistert siden datamaskinenes barndom og er fortsatt blant de aller vanligste elektroniske lagringsmedia. I de siste årene har Solid-State Drives (SSD-er også hatt betydelig suksess. Som mekaniske innretninger er disker flere størrelsesordner langsommere enn solid-state-teknologier. Mens SSD-er fungerer på mikrosekundskalaen, bruker disker typisk flere millisekunder på å gjennomføre en lese-/skriveoperasjon. Flere faktorer bidrar til forsinkelsen, aller viktigst søkeforsinkelse, rotasjonsforsinkelse og overføringstid.

Gitt en 7200 RPM disk, hva er rotasjonsforsinkelsen og hvordan kan den utledes? Videre, gitt at disken har gjennomsnittlig 617 sektorer per spor og 512 bytes blokkstørrelse, hva er overføringsraten?

Rotasjonsforsinkelsen er den tiden det tar for disk platene å rotere slik at den ønskede sektoren er under diskhodet. For å finne rotasjonsforsinkelsen må vi først finne hvor mange millisekunder det tar for disken å spinne en runde. Da må vi dele 600000 ms på 7200rpm.
60000ms/7200rpm = 8,3333333333ms/r

Siden gjennomsnittlig forsinkelse er på en halv runde må vi dele 8,3333333333 på to. Da får vi at gjennomsnittlig forsinkelse på en disk med 7200 rpm er omtrent 4,17 ms.
Overføringsraten finner vi slik:

617 sectors × 512 bytes = 315904 bytes

315904 bytes/1024/1024 = 0.301269531 MB

(7200/60) × 0.301269531 MB = 36,152 MB/s

Altså er overføringsraten på 36,152 MB/s.

Svaret under er feil, men har riktige benevnelser. Svaret over er riktig, men har ikke riktige benevnelser. En kombinasjon av begge besvarelsene, ville da ha vært ideelt. Så hvis svaret under hadde vært riktig så hadde besvarelsen vært perfekt.

Gitt en 7200 RPM disk, hva er rotasjonsforsinkelsen og hvordan kan den utledes? Videre, gitt at disken har gjennomsnittlig 617 sektorer per spor og 512 bytes blokkstørrelse, hva er overføringsraten? Den gjennomsnittlige rotasjonsforsinkelsen er tiden det tar for en halv rotasjon. Dette er fordi blokken som skal hentes kan ligge hvor som helst på disken, og gjennomsnittet blir på midten av disken. For en 7200 RPM disk blir dette 4,17 ms.
Overføringsraten regnes ut ved å dele antall sektorer per spor på tid per rotasjon. Hvis vi må endre spor, må også tid legges til for å flytte hodet.

$$\frac{\frac{512\ bytes \times 617\ sektorer}{1000\ m/s \times 60\ sec/min}}{7200\ RPM} = \frac{316\,904\ bytes\ per\ track}{8{,}33\ ms} = 38\,044\ b/ms = 38\ MB/s$$

I dette tilfellet får vi omtrent 38 MB/s.

Diskkontrolleren dirigerer de mekaniske operasjonene som leser og skriver data til og fra platene. Gi en sammenfatning av leseprosessen med fokus på det mekaniske aspektet.

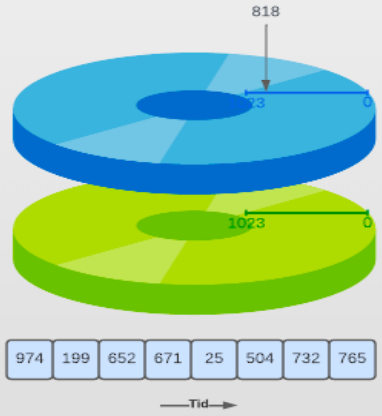
Diskkontrolleren kontrollerer aktuatoren som beveger diskhodet til det ønskede sporet i disken. Tiden det tar å komme til ønsket spor kalles søketid (seek time). Den venter så til platen har rotert så mye at det første bitet som skal has tak i passerer under diskhodet. Tiden ventinga tar kalles for rotasjonsforsinkelsen (rotational delay). Så kontrolleres overføring av data mellom hovedminnet og disken. Denne overføringen er ferdig når alle bitene man ønsker å lese har passert under diskhodet. Tiden dette tar kalles overføringstiden (transfer time). I tillegg kan det være andre faktorer som påvirker den totale aksesstiden, for eksempel ventetiden på at CPU-en skal prosessere forespørsel fra bruker.

Skedulering

Norsk

Diskoperasjoner er notorisk trege. De vil typisk utgjøre en ytelsesflaskehals for enhver prosess som er avhengig av dem. Lese-og-skrive-hodets mekaniske bevegelse medfører en tidsstraff som hovedminnet (samt *solid state drives* (SSD)) unngår ved å ha ingen bevegelige deler. Straffen er streng: Diskoperasjoner er omtrent seks størrelsesordener langsommere enn minneoperasjoner. Hvor effektivt – eller ineffektivt – diskoperasjoner skeduleres påvirker av denne grunnen ytelsen til alle I/O-bundne prosesser; dermed også systemets ytelse i sin helhet. Det finnes flere diskskeduleringsalgoritmer. Vi vil se på tre av dem her: *First Come, First Serve* (FCFS), *Shortest Seek Time First* (SSTF) og SCAN.

Se på diskanatomien og skeduleringskøen vist i Figur 2. Disken har 1024 sylindre, nummerert fra 0 nærmest kanten til 1023 nærmest sentrum.



Figur 2: Diskplanlegging

Når vi velger skeduleringsalgoritme, hvilke mål ønsker vi å oppnå?

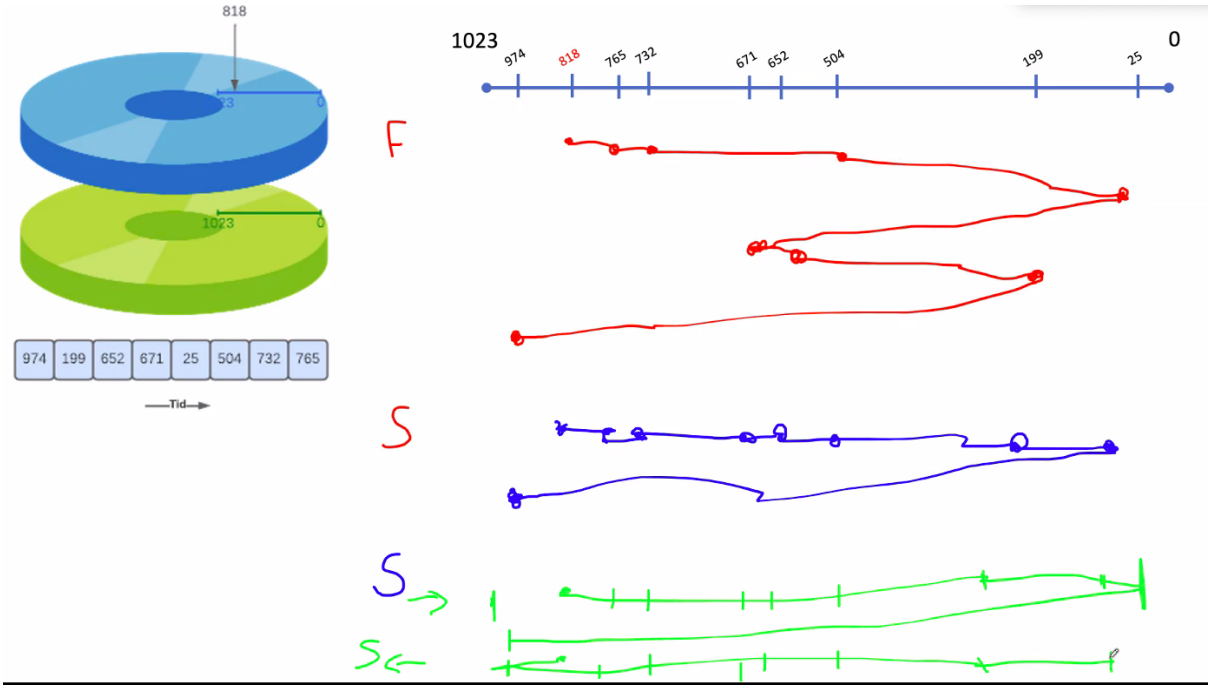
Vi ønsker å oppnå kortest mulig responstid, altså at hver innkommende forespørsel om minneallokering/henting blir foretatt så fort som mulig. Vi ønsker også høy throughput, altså at tiden fra en forespørsel blir sendt, til den blir ferdig utført, blir så kort som mulig. Til slutt ønsker vi også rettferdighet, at de forespurte minneoperasjonene blir utsatt for rettferdige delays.

I denne besvarelsen, så er det riktige punkter som er viktig for alle skeduleringer. Men vi må også se fra disken sitt syn hvor vi ser på hvordan vi håndterer disken som en ressurs. F.eks så vet vi at disken har en diskhode og vi ønsker f.eks at skeduleren skal være bygd slik at diskhodet beveger seg minst mulig. Altså at søkeraten for oppslag av en gitt spor, er kort for diskhodet, og dette er viktig å inkludere. Dette glemte mange av gruppene å nevne hvor de kun tok for seg generelle mål vi ønsker å oppnå ved skedulerere som gjelder for både CPU og Disk. Problemet var mangel på å begrunne målene vi ønskers spesifikt ved en Disk-skeduler. Da er søkeraten og rekkefølgen om å utføre diverse datablokker viktig å forholde seg til ved en disk-skeduler.

Med tanke på søketid, hvordan sammenliknes FCFS, SSTF og SCAN i dette scenarioet?

Bildet under, beskriver Pål sitt bildet av sammenlikning av de ulike algoritmene. Han antok at 765, var den første blokken som skulle utføres, så 732, 504 osv. “Rød F” står for “FCFS”, “Rød S” står for “SSTF” og “Blå S” står for “SCAN”. Han gjorde en antakelse ved “SCAN”

hvor han da lagde to antakelser hvor vi må ta hensyn til at diskhodet kan gå fra akselen og mot ytterkanten, eller at diskhodet går fra ytterkanten og mot aksen. Det har han vist helt nederst, hvor den “Blå S” referer til tilfellet hvor diskhodet går mot ytterkanten og tilbake til aksen, den “Grønne S” referer til tilfellet hvor diskhodet går mot aksen også til ytterkanten.



Vi ser at Scan er den beste av de oppgitte algoritmenen, så er det SSTF og tilslutt FCFS.

Bortsett fra søketid, yter algoritmene forskjellig etter andre ytelseskriterier? I tilfelle hvilke og hvordan?

Andre ytelseskriterier kan inngå “starvation”, hvor “FCFS” og “SSTF” kan ha en del starvation. I “FCFS” så kan det hende at prosesser som vi ser i bildet over, må vente en god stund før de blir utført. F.eks så måtte “974” vente lenge på å bli utført siden den er sist i køen. For “SSTF” så kan det også forekomme “starvation” ettersom vi kan ha mange prosesser i aksen og kun en prosess helt ytterst mot kanten av platene. Denne prosessen må vente lenge ettersom “SSTF” lar prosessene mot aksen bli utført først.

“FCFS” er lettere å implementere enn alle andre algoritmer med tanke på å konstruere algoritmen. Dette skyldes av at det er et ganske simpel prinsipp som benyttes ved denne algoritmen sammenlignet med andre.

Man kan også tenke på aksesstiden ved disk, hvor algoritmer som har lavere søketid er ideelt for å unngå at aksesstiden blir lang med tanke på å rotasjonsdelay og overførelsesraten for prosessene. Derfor er “SCAN” ideelt å benytte med tanke på at aksesstiden ved diverse datablokker blir mindre enn andre algoritmer og vi har grafen over som et eksempel på dette hvor SCAN har mindre tid enn ved de andre.