Prosjektøving eksamenstrening

NB: Du skal ikke løse oppgave 1 b) og c) samt oppgave 5 c) og d) da dette er tema dere får etter jul.

**Oppgave 1. Tallsystemer og datarepresentasjon**

Heksadesimalt

Når jeg skal konvertere til heksadesimal form vil jeg begynne med å dele opp bitmønstrene i 4 bits hver. Heksa betyr 16 eller 2^4, derfor kan man enkelt regne om den binære verdien til den heksadesimale verdien med 4 og 4 bits. Deretter legges alle svarene i heksadesimalt sammen. Det er viktig å legge sammen i riktig rekkefølge, jeg starter fra venstre og går mot høyre.

De ulike bitmønstrene man får med 4 og 4 bits, må først konverteres til desimal for å så konverteres til heksadesimalt. De heksadesimale tallene er like tallene på desimal form opptil 9, etter dette brukes bokstavverdier: F.eks. desimal form: 10, heksadesimal form: A osv… til 16.

Den heksadesimale rekkefølgen er som følger: 0 1 2 3 4 5 6 7 8 9 A B C D E F

Binært

Når jeg skal konvertere til desimal form vil jeg bruke de binære verdiene oppover n antall bits opphøyd i 2, altså 2^n. De binære verdiene er som følger fra høyre til venstre side: 1 2 4 8 16 32 64 128. Man kan se at disse verdiene er 2^n, hvor n er hvilken bit man befinner seg på. F.eks. 101 blir 2^0\*1 + 2^1\*0 + 2^2\*1 = 3.

Legg merke til at det ikke er nødvendig å plusse på tall hvis den binære verdien er 0, siden svaret uansett blir 0 når man ganger med 0, men det kan skrives for oversiktens skyld. Det samme gjelder hvis det en bit er 1. På grunnlag av dette kan man regne alle de desimale verdiene sammen og få dermed få det på desimal form.

**Bitmønster A: 00010111**

Heksa: 0001 = 1 og 0111 = 7

Heksadesimal verdi = 1 + 7 = 17

Desimal: 2^0 + 2^1 + 2^2 + 2^4 = 1 + 2 + 4 + 16 = 23

**Bitmønster B: 10101110**

Heksa: 1010 = A og 1110 = E

Heksadesimal verdi = AD

Desimal: 2^1 + 2^2 + 2^3 + 2^5 + 2^7 = 2 + 4 + 8 + 32 + 128 = 174

**Oppgave 2. Busser**

De tre hovedtypene av busser som er omtalt i kurset er: *parallell buss, seriell buss* og *svitsjet buss*.

1. **Beskriv oppbyggingen til hver av disse hovedtypene.**

**Parallell buss**: en parallell buss er en buss som har mer enn en tilkobling for overføring av data. Dataene kan sendes over bussen parallelt med hverandre, altså en «kabel/tråd», kan overføre noe data, samtidig som en annen overfører noe annen data. Bussbredden på en parallell buss sier hvor mange bits som kan overføres samtidig. Den vanligste typen parallell buss er synkrone, det vil si at de kan kun sende signaler ved klokkepulser. Antall pulser hvert sekund bestemmes av klokkefrekvensen, og måles i Hz.

En parallell buss består hovedsakelig av data-, adresse- og kontroll buss. Adressebussen skal fortelle hvilken komponent som skal motta dataene. Databussen overfører de binære dataene dit den skal. Kontrollbussen overfører signaler som styrer og synkroniserer overføringen.

**Seriell buss**: en seriell buss er en buss som kun har en tilkobling i hver ende for overføring av data. Det vil si at data kan overføres etter hverandre. Ikke parallelt som med den parallelle bussen. Altså adressene og dataene sendes over samme buss sekvensielt. Man skiller gjerne mellom tre forskjellige typer seriell buss:

*Simpleks*: kommunikasjon som er enveis, altså signalet kan kun gå en av veiene.

*Halv-dupleks*: kommunikasjon kan foregå begge veier, men ikke samtidig.

*Full-dupleks*: kommunikasjon kan gå begge veier, samtidig.

**Svitsjet buss**: en svitsjet buss har muligheten til å koble sammen komponenter på parvis basis. En svitsjet buss kan koble opp flere tilkoblinger, og hver tilkobling vil være et punkt til punkt forbindelse. Denne fungerer på samme måte som en telefonsentral. Selve svitsjen vil være en sentral som bestemmer hva som skal kobles sammen. Det brukes som oftest seriell buss for denne tilkoblingen.

1. **Nevn fordeler og ulemper med hver av disse hovedtypene.**

**Parallell buss:**

Fordelen med parallelle busser er den raske overføringshastigheten over korte avstander. Man kan sende alle bitene som trengs i en overføring på en klokkepuls. Dette medfører også enklere elektronikk, siden bits sendes uforandret over bussen.

Det finnes begrensinger til den optimale fysiske størrelsen på en parallell buss, samt hvor mye man kan skru opp klokkefrekvensen før det oppstår feil i kommunikasjonen. Det er ikke ønskelig å øke bredden av parallelle busser mer enn det som finnes i dag grunnet praktiske hensyn. I tillegg hvis man skrur opp klokkefrekvensen vil det oppstå små forskjeller på når signalene ankommer enden av en buss eller motsatt. Altså det er forskjeller i tiden de elektriske signalene bruker gjennom bussen grunnet at de kan bli påvirket av omgivelsene. Dette vil medføre at noen signaler kommer før eller senere enn andre. Prosessoren opererer i nanosekunder, derfor er timing veldig viktig. I tillegg er det begrensninger til lengden på en parallell buss.

**Seriell buss**: En seriell buss kan oppnå høy båndbredde over lengre distanser. Man kan skru opp klokkefrekvensen meget høyt, siden man ikke får problemer med elektriske forstyrrelser på serielle busser. Da gitt at det ikke finnes noen eksterne kilder som gir støy. Dimensjonene på utstyret er også små, som gir praktiske fordeler. Ulempen med serielle busser er at det kreves mer avansert elektronikk/kontrollere. Dessuten så dukker spørsmålet om hvilken rekkefølge bitene skal sendes i opp.

**Svitsjet buss**: Fordelen med svitsjet buss er at flere enheter kan kommunisere med hverandre samtidig. I tillegg kan man tildele flere tilkoblinger til enheter som har behov for det. Dette gjør at man kan oppnå en høyere båndbredde en med kun en tilkobling. Ulempen med dette er at det kreves mer avansert elektronikk/kontrollere for å kontrollere en svitsjet buss. Man må blant annet ha en eller annen form for adressering for å skille mellom enhetene på bussen. Dessuten kan det være vanskelig å bestemme hvem som skal få flere tilkoblinger. Dette løses med standarder som PCI Express.

1. **Moderne datamaskiner bruker gjerne alle disse hovedtypene av busser. Nevn minst et eksempel på moderne busser fra hver av hovedtypene.**

**Parallell buss**: Bussen mellom prosessoren og primærminnet på hovedkortet til en datamaskin er et eksempel på en parallell buss. Man ønsker å ha kort avstand mellom komponentene med en parallell buss, noe som er grunnen til at primærminnet ligger så nært prosessoren.

**Seriell buss**: USB (Universal Serial Bus) er et moderne eksempel på en seriell bus. USB er en ekstern buss, ikke som kobberbaner på kretskort, det er grunnen til at det benytter seg av seriell bus teknologi siden de andre krever større plass. USB er en standard som er mye brukt, spesielt til tilkobling av elektroniske enheter. SATA er også en standard som tar i bruk seriell buss. SATA brukes ofte for kommunikasjon med lagring som harddisker eller disk.

**Svitsjet buss**: Standarden PCI-Express er et eksempel på en svitsjet buss. Denne bussen erstatter den eldre teknologien PCI. Den store fordelen med PCI Express over PCI er at man kan bruke flere «lanes» til samme enhet for høyere båndbredde. Altså man kan sette opp flere tilkoblinger til en enhet hvis det er tilkoblinger ledig. Dette gjør at kommunikasjon mellom enheter på bussen skjer mye raskere. PCI Express er ofte brukt på hovedkort til datamaskiner for å tilkoble eksterne utvidelseskort. F.eks. grafikkort er et utvidelseskort som ofte krever kjapp kommunikasjon. PCI Express fungerer derfor som hovedbussen på moderne pc-er.

Oppgave 3. Cache og Systemarkitektur

1. **Hva sier prinsippet om lokalitet? Begrunn hvorfor prinsippet gjelder.**

Prinsippet om lokalitet sier at ting som ligger nært hverandre vil påvirke hverandre. Altså hvis man må hente data i et lager som bruker lang tid på å finne og overføre data, så kan man ta en «blokk» med data og lagre i et raskere og mindre «mellomlager», også kalt cache. Det er stor sannsynlighet for at data som ligger nært den dataen som vi må ut f.eks. i primærminnet å hente, kommer til å bli brukt i nær fremtid.

For eksempel hvis et program tar i bruk en for loop, vil den bruke data i for loopen flere ganger, derfor er det mer effektivt å lagre denne dataen i cachen, enn å måtte hente den fra primærminnet hver gang. Dette gjør prosesseringen av oppgaven raskere, siden man slipper å vente på IO eller andre lagre som primærminnet.

Dette prinsippet gjelder ikke bare for primær minnet og cachen, men for alle enhetene i lagerhierarkiet. F.eks. de fleste harddisker har en liten buffer som skal gjøre prosessen med å overføre data mer effektiv.

1. **L1-cachen til en prosessor oppgis til å være 8-veis sett-assosiativ cache med linjestørrelse på 64 byte. Forklar nøye begrepene som er understreket, og hvorfor det ofte er en hensiktsmessig oppbygging.**

For at en cache skal være effektiv har CPU kun noen nanosekunder på å sjekke om data finnes i cachen eller ikke, derfor har hvordan blokker lagres i cachen mye å si. En cache kan ha tre ulike mapping funskjoner:

**Full-assosiativ**: en blokk med data kan legges i hvilken som helst linje i cachen. Dette betyr at når CPU skal sjekke etter data i cachen, må den sjekke gjennom hver linje til den finner den riktige dataen, eller alle linjene dersom den ikke finnes i cachen. Dette er tidskrevende og lite effektivt for cachen, selv med den hurtigste elektronikken som finnes i dag.

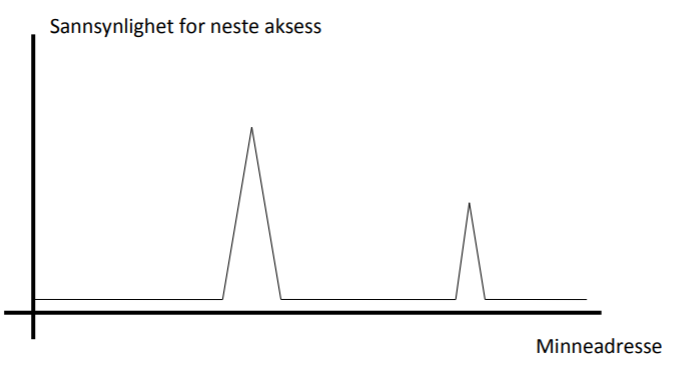
**Ikke-assosiativ (direkte mapping):** en blokk «tilhører» en linje i cachen. F.eks. linje 0 vil kun inneholde blokk 1, 3 eller 5, og linje 2 vil kun inneholde 2, 4 eller 6, da gitt at en av disse blokkene er i cachen. Da trenger CPU kun å sjekke den spesifikke linjen til blokken den ønsker å finne. Det er også viktig å merke seg at en cache har mye flere linjer enn beskrevet.

Ulempen med denne mapping-funskjonen er trashing. Hvis to data-verdier som har samme tilhørighet, altså kan ligge i samme linje, vil de tvinge hverandre ut av cachen, gitt at de brukes i instruksjoner etter hverandre. Trashing vil medføre at cachen blir mindre effektiv, og noen ganger sinker systemet.

**Sett-assosiativ:** en blokk tilhører en linje i cachen, på lik måte som i ikke-assosiativ. Derimot så skal sett-assosiativ løse problemet med trashing. Siden blokker konkurrerer om å ligge i linjen, kan man legge flere linjer i et sett. Altså flere blokker som har samme tilhørighet kan ligge i samme sett. Det finnes ulike typer sett, 2-veis vil ha 2 linjer per sett, 4-veis vil ha 4 linjer per sett, og 8-veis vil ha 8 linjer per sett.

**Linjestørrelse**: antall byte i linjestørrelsen sier hvor stor blokk som kan ligge på hver linje. CPU vil også bruke denne informasjonen til å finne ut hvilken blokk den skal se etter. F.eks. med en linjestørrelse på 64 byte, vil den første blokken bestå av adressene 0 – 31, den andre 32-63 også videre. Da vet f.eks. CPU at adresse 23 vil befinne seg i blokk 0, og kan bruke denne informasjonen til å sjekke om blokken ligger i cachen eller ikke.

1. **Dersom man tar et normalt program og ser hvilke minnelokasjoner som i praksis brukes av programmet, vil man se at det bruker to områder i minnet. Dette er skjematisk fremstilt i figuren nedenfor. Plasseringen av disse områdene i minnet vil bestemmes av operativsystemet**



Svar på følgende spørsmål:

* + **Hvorfor er det to slike områder?**

Prosesser tildeles et eget område i minnet til både instruksjoner og data. Derfor vil man se to slike områder. Prinsippet om lokalitet vil gjelde for disse områdene. Altså sannsynligheten er stor for at neste instruksjon eller data som trengs, ligger i disse områdene.

* + **Har denne egenskapen med programmene noe å si for at sett-assosiativ cache er den vanligste mapping-funksjonen?**

Hvis adresser i et begrenset område er mest brukt, vil det si at blokker kan tvinge hverandre ut hvis det kun er plass til en blokk per linje. Derimot hvis man bruker sett-assosiativ cache vil den sørge for at flere blokker kan ligge i samme sett. Derfor kan man si at sett-assosiativ er den vanligste mapping-funksjonen.

1. **Moderne minneteknologier er spesiallaget for å utføre såkalte «burst».** 
   * **Hva er en «burst»?**

Den parallelle bussen som ligger mellom cachen og minnet er som regel 64 bits, eller 8 bytes. Med en linjestørrelse på 64 byte som i oppgave b), må man overføre 8 ganger over bussen for at alle dataene skal bli overført. Ved den første overføringen vil man måtte vente noen klokkepulser siden minne-elektronikken må finne fram rett lokasjon for dataene, altså der dataene skal ligge i cachen. Derimot når denne lokasjonen er funnet, kan minnet sende data på hver kontinuerlig klokkepuls ved å bruke lokasjonene fra startadressen og utover. Det er dette som kalles burst. I moderne DDR-SDRAM brukes «double data rate», som vil si at det overføres to ganger per klokkesyklus.

Kort sagt så er en burst når man kan kontinuerlig sende over data på bussen etter at lokasjon for start-adressen er funnet.

* + **Med utgangspunkt i det vi har lært om systemarkitektur: Forklar detaljert hvordan en «burst» utføres.**

En burst kan beskrives i tre hovedsteg:

* + 1. **Overføring av adresse til minnet**: For at minnet skal vite hva som skal overføres, blir en adresse sendt over bussen. Denne adressen som overføres helt i starten av overføringen blir betraktet som startadressen til blokken.

* + 1. **Vente til minnet finner rett lokasjon**: Når denne startadressen er overført må minnet finne fram rett lokasjon til denne adressen. Dette vil være aksesstiden til minnet. Legg merke til at klokkefrekvensen ikke har noe å si for denne aksesstiden, men akkurat i det minnet er ferdig må vi vente til neste klokkepuls.
    2. **Kontinuerlig overføring**: Etter at lokasjon i minnet er funnet, kan minnet begynne å sende ut data på hver klokkesyklus. Det vil si at ved hver gang en ny klokkesyklus oppstår, så vil det sendes data på bussen. Da vil informasjonen som sendes over bussen lagres i en linje i cachen.

Derfor kan man si at det går litt tregt i starten, men etter det blir det overført kontinuerlig/fort. Legg merke til at i dette tilfellet har klokkefrekvensen til bussen mye å si for tiden det tar å overføre, siden vi må vente på at en ny klokkepuls for hver overføring. Det er også derfor det er ønskelig å øke klokkefrekvensen så mye som mulig uten at det oppstår praktiske restriksjoner.

**Oppgave 4. Prosesser og tråder**

1. **Vi snakker ofte om både prosesser og tråder. Forklar hva en prosess er og hva en tråd er og hvilken sammenheng det er mellom prosesser og tråder.**

Prosesser er program kode som kjører. Man kan se på program koden som ligger på en kjørbar fil som «oppskriften» på en prosess, altså prosessen vil følge denne «oppskriften». Prosesser har et eget område i minnet som kun er avhold til en spesifikk prosess.

Tråder derimot er subprosesser. Man kan se på tråder som prosesser inni prosesser. En prosess har en eller flere tråder. Alle trådene deler på instruksjonsområdet til prosesoren, men har sin egen stack og dataområde. Sammenhengen mellom prosesser og tråder er at tråder oppstår inni en prosess, en tråd vil alltid tilhøre en prosess

Prosesser har en prosessdeskriptor som inneholder nødvendig informasjon om prosessen som trengs ved kjøring. Blant annet så lagres id (PID, status om kjøring, registre etc. Ved suspendering av en prosess trengs denne prosessdeskriptoren for å vite hvor prosessen var i kjøringen når den ble suspendert.

OS vil gi en illusjon av at man har flere kjerner enn i virkeligheten ved hjelp av disse trådene, dette kalles «thread abstraction» For brukeren vil det virke som at alle oppgavene som kjøres på prosessoren gjøres samtidig/parallelt. F.eks. bruker laster ned en fil, samtidig som brukeren hører på musikk og beveger på musen på skjermen. OS bestemmer hvor lenge en tråd skal få kjøre på prosessoren før den bytter til neste. Prosessorene i dag er såpass kjappe at de vil gi denne illusjonen av at alt gjøres parallelt, gitt at man begrenser antall tråder.

1. **Hvordan er det mulig å kjøre flere tråder på en CPU enn det er tilgjengelige kjerner i CPU-en? Forklar.**

En prosess vil være det som kjører i en kjerne på en CPU. Inni denne prosessen kan det finnes flere tråder som hele tiden vil bytte på å kjøre. Det finnes grenser for hvor mange tråder man kan ha inni en prosess, men det bestemmes ikke av antall kjerner i CPU-en. Begrensingen for antall tråder bestemmes av den fysiske maskinvaren eller operativsystemet. Altså man kan kjøre prosesser på kjernene i en CPU, og inni disse prosessene vil det være flere tråder som kjøres etter hverandre.

OS har en schecdueler i prosessadministrator som bestemmer hvor lenge disse trådene skal få kjøre. Dette kan være et tidsintervall hvor hver tråd får en gitt tid på å kjøre før den bli «kastet» ut igjen. Dette gir illusjonen at flere tråder kjører samtidig.

1. **Tjenerprogrammer, f.eks på en filtjener eller på en webtjener, er som oftest såkalt ”multithreaded”, dvs kjører mange tråder. Hvorfor er det hensiktsmessig å kjøre mange tråder på f.eks en filtjener sammenlignet med å kjøre kun en prosess? Forklar.**

En fil- eller webtjener vil få masse forespørsler hele tiden. Hvis denne fil- eller webtjeneren måtte vente på at en forespørsel skal bli ferdig behandlet før den kan starte med den neste, vil tjeneren virke treg for brukerne. Derfor kan man ta i bruk tråder. Hver forespørsel kan få sin egen tråd som kan arbeide med forespørselen samtidig med de andre trådene. Dette vil gi en mye mer effektiv behandling av forespørsler, som er viktig for en fil- eller webtjener.

1. **Prosesser får tilgang til i/o-enheter via operativsystemkjernen. På hvilken måte? Forklar.**

For at prosesser skal få tilgang til IO enheter må den sende ut et systemkall til os-kjernen. Dette må gjøres fordi prosesser kjører i bruker-modus. Aksessering av IO enheter kan kun gjøres av kjernen for sikkerhetsmessige grunner, derfor må prosessene som er i bruker-modus spørre kjernen om den kan aksessere IO enheten med et systemkall. I OS-kjernen er det prosedyrer som sørger for at de som kaller systemkallet ikke har utnyttet seg av noe for å utføre kommandoer som de egentlig ikke har tilgang til.

Hvis prosessen har rettigheter for å utføre etterspurt IO vil et systemkall jobbe mot driverprogrammer og foreta IO. OS vil da samhandle med IO og på prosessoren vegne.

1. **Hvorfor kan ikke prosessene gå direkte mot i/o-enhetene og gjøre sin egen i/o uavhengig av operativsystemkjernen? Forklar.**

Det er av sikkerhetsmessige årsaker at prosessene ikke kan gå direkte ut mot IO-enhetene, og derfor må spørre OS-kjernen om å utføre det for dem. Hvis man hadde tillatt prosesser å aksessere IO direkte uavhengig av OS-kjernen ville det ha gått imot et av hovedprinsippene som operativsystemet bygger på, altså sikre brukerens data, maskinvare og sørge for at systemet i sin helhet kjører stabilt og som det skal.

La oss f.eks. si at prosesser fikk direkte tilgang til IO. Da har de muligheten til å aksessere informasjon i minnet utenfor prosessens gitte adresseområde. Altså informasjon som prosesser ikke skal ha tilgang til kan bli lest. I tillegg kan ikke operativsystemet sikre at prosessene utfører de riktige kommandoene i riktig rekkefølge for å f.eks. lese fra en disk. Alt dette i sin helhet vil medføre store sikkerhetsbrudd og risikere stabiliteten til systemet.

Man antar alltid at applikasjoner/prosesser ikke er til å stole på. Derimot OS-kjernen kan man stole på siden koden i denne er utviklet av «oss» eller noen man stoler på. Koden i applikasjonene kan være skadelig, og f.eks. inneholde virus. Man kan tenke at applikasjonene som kjører i bruker-modus ligger inni en lukket «boks» hvor de kan gjøre hva de vil, men hvis de vil gjøre noe utenfor «boksen» må de spørre OS-kjernen. All aktivitet inni denne «boksen» kan ikke føre til sikkerhetsbrudd i systemet.

**Oppgave 5. Minne**

1. **I minneadministrasjon snakker man om relokering av adresser som betyr omregning fra virtuelle adresser til fysiske adresser. Forklar hvorfor man trenger denne omregningen fra virtuelle til fysiske adresser. Forklar også hvordan denne omregningen foregår i dagens operativsystemer som f.eks i Windows eller Linux (ikke tekniske detaljer her, kun prinsipper om hvordan det foregår).**

Alle prosesser har et eget virtuelt minne, slik at programmet ikke ser noen begrensinger i den underliggende maskinvaren og slik at man slipper å kompilere programmet hver gang det flyttes i minnet. Et virtuelt minne vil starte på adressen 0 og utvides utover dette. Prosessoren bruker adresser til dette virtuelle minnet når den kjører programmet. Disse virtuelle adressene tilsvarer ikke til de fysiske adressene i minnet, dessuten kan to forskjellige programmer bruke samme virtuelle adresse på et tidspunkt. Løsningen på dette er å konvertere fra virtuelle adresser til fysiske adresser som kan brukes i minnet.

Denne konvertering foregår i MMU (Memory Management Unit) plassert mellom CPU og minnebussen. Altså det er MMU som regner om de virtuelle adressene til de fysiske adressene til minnet, de virtuelle adressene sendes aldri ut på minnebussen.

1. **Du har en datamaskin med 64 bits CPU. Fysisk minne er på 16 Gbyte. Men hvor stort er det virtuelle minnet? Jeg er ikke ute etter et eksakt tall her, men en diskusjon rundt temaet, f.eks vil jeg vite noe om både hvor stort virtuelt minne operativsystemet kan sette av og hvor mye virtuelt minne hver prosess disponerer.**

Teoretisk størrelse på et minne med 64 bits vil være 2^64, som er mye mer enn 16GB. Det virtuelle minnet ligger derimot ikke kun i det fysiske minnet, men også på mellomlageret på harddisken. Derfor kan det virtuelle minnet i praksis være mye større enn det fysiske minnet, i tillegg til den teoretiske størrelsen på 2^64. Til tross for dette så er som oftest ikke praktisk anvendelig å ha slike store virtuelle minner.

Hver prosess har bare et virtuelt minne som vokser så stort som det trenger å bli. Når prosessen kjører vil kun de delene som er nødvendige være i det fysiske minnet (RAM). I tillegg når man lager et program så trenger man ikke å ta like stort hensyn til minnebruk. På grunnlag av dette kan man si at prosessene er mer fleksible med virtuelt minne.

1. **Forklar hva sidedelt minneadministrasjon går ut på og redegjør for hvilke fordeler det er med sidedelt minneadministrasjon.**

En sidedelt minneadministrasjon deler opp det virtuelle minnet inn i like deler kalt sider. Størrelsen på sidene kan variere, men de er som regel ikke store, typisk størrelse i systemer i dag er 4KByte.

Når det virtuelle minnet deles opp i like store blokker som det fysiske minnet, kan man enkelt flytte over en blokk fra det virtuelle minnet inn i en blokk i det fysiske minnet. Da slipper man å lete etter ledige plasser i det fysiske minnet. Det gir også fordelen med at sidedelte systemet kraftig reduserer behovet for defragmentering/kompaktering.

1. **Baseregister og sidetabell har sentrale funksjoner i minneadministrasjon. Forklar hvilken funksjon de har og hva som skiller og hva som er felles for baseregister og sidetabell.**

Baseregisteret skal angi startadressen til programmet i minnet. Man bruker baseregisteret for å oversette de virtuelle adressene til fysiske minne-adresser. Ved å ha et baseregister kan man enkelt endre baseregisteret hvis programmet blir flyttet i minnet. Dette kan forekomme hvis det oppstår «hull» i minnet og det må defragmenteres/kompakteres. Denne prosessen er tungt for OS så den vil helst unngå det. Derimot de fleste nye operativsystemer har teknikker for å unngå dette derfor er det som regel ikke et problem. I tillegg kan det forekomme hvis et program blir byttet av et annet program og deretter blir lagt til igjen i minnet på en annen lokasjon, noe som skjer ofte når minnet fylles opp. Baseregisteret befinner seg i prosessdeskriptoren til prosessen. Prosessene har mange slike baseregistre, derfor brukes sidetabeller.

Baseregistre brukes til å angi hvor langt unna adresse 0 adressen er i minnet i forhold til i det virtuelle minnet. Derimot sidetabeller brukes til å holde styr på hvor blokkene ligger i minnet, og om de i det hele tatt ligger i minnet. Ved å bruke både baseregistre og sidetabeller kan man regne om fra virtuelle adresser til fysiske adresser.

Den virtuelle adressen som sendes til MMU består av sidenummer og offset/baseregister. Antall bits som brukes for hver del av adressen vil angi antall sider og sidestørrelsen. En sidetabell består av sidenr, blokknr og en kolonne med navnet «tilstede». Når den virtuelle adressen kommer til MMU, vil den finne tilsvarende sidenummer i tabellen og hvilket blokknummer som ligger i denne rekken. Deretter vil den multiplisere blokknummeret med side-/blokkstørrelsen og plusse på baseregisteret for å få riktig adresse i minnet. I tillegg vil dette bli multiplisert med tilstede-verdien, hvis den er 1 vil man få samme adresse. Derimot hvis den er 0 vil man få adresse 0 som tilsvarer NULL siden blokken ikke finnes i minnet. Da vil operativsystemet avbryte kjøringen av prosessen til blokken er hentet fra disk.