Minne

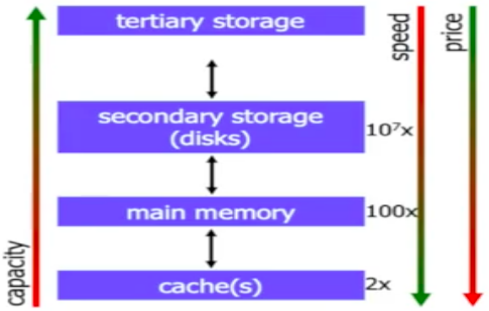
Minnehåndtering

Ved minnehåndtering så er det viktig at man håndterer systemets minne ressurser. Blant annet er det viktig å allokere plass til en prosess, og beskytte deres minneplass. Allokering i denne sammenhengen vil si at vi gir prosessen et gitt minneplass. Derfor er det viktig å beskytte minneplassen som ble allokert for en prosess.

Det man prøver å oppnå ved minnehåndtering, er å håndtere de ulike nivåene av minne, fordele de i forskjellige prosesser og beskytte de prosessene fra hverandre. La oss si at vi har en prosess “A” og prosess “B”, vi skal sørge for at disse prosessene har sin egen minneområde, ikke slik at a “A” kan lese minne til “B” osv.

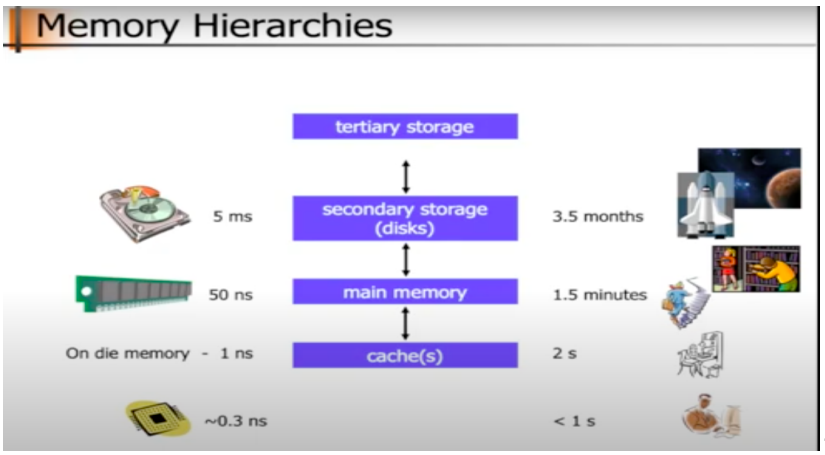
Minnehierarki

En prosess trenger som oftest mye minne, men hastigheten til overføring av data mellom disk og prosess, kan ta mye tid. Derfor er ikke RAM, eller selve harddisken på maskinen en god løsning for en prosess å aksessere minne fra. Dette husker vi fra IN1020 hvor “Register” og “Cache” har raskere minneaksessering enn RAM og harddisk. Så under skal vi da se mer om minnehierarki og lære mer om dem.



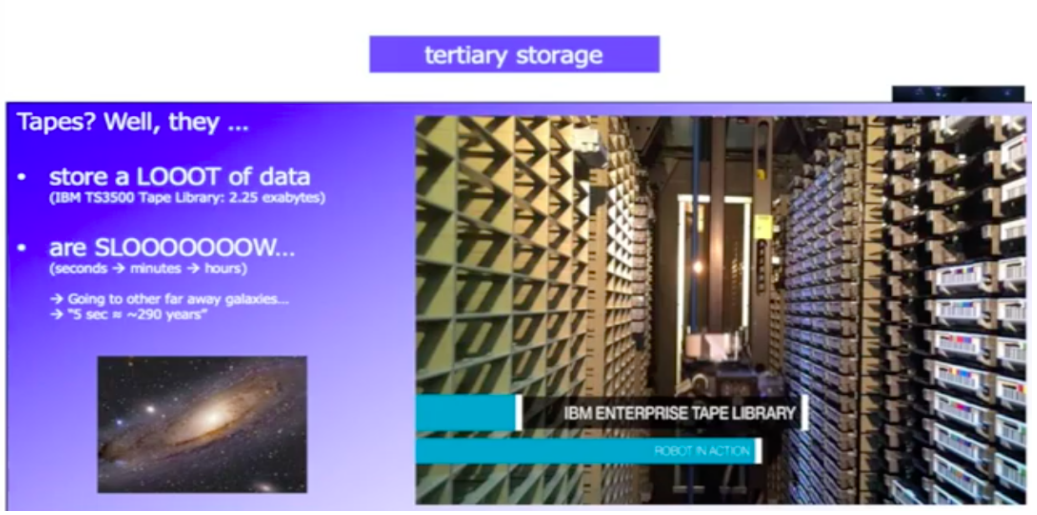
Register

Her ser vi et eksempel på et minnehierarki hvor “Cache” er helt nederst(Merk register er helt i bunnen. Deretter primærminne er over hvor vi har da RAM-brikkene i dette primærminne og tilslutt så har vi harddisk. Grunnen til hvorfor vi har dette hierarkiet, skyldes av at register og cache, kan ikke oppbevare alt av minne. Disse for en kopi av dataen som ligger i disken som gjør at aksesstiden for å hente data fra register og cache, er raskere enn å hente fra disken. Så derfor vil primærminne kunne oppbevare mer minne enn register og cache. Tilslutt vil harddisken oppbevare mest mulig minne. Bildet over viser altså at de lavere nivåene har bedre fart enn de høyere nivåene, men har dårligere kapasitet sammenlignet med disk osv.



<-------(**REGISTER)**

Bildet over beskriver hvor lang tid det tar for CPU´en å aksessere data mellom de ulike minnene vi har i bildet. **NB!** Helt nederst så er ikke registeret blitt inkludert i bildet, men det tar en klokkesykkel å aksessere minnet herifra, altså 0,3 nanosekunder. Vi husker fra IN1020 at Cache er delt opp i nivåer hvor vi har L1 Cache, L2 Cache osv. Den nærmeste altså L1 Cache, vil ta 1 nanosekund. Primærminnet vil ha typiske tider ha 30,40 eller 50 nanosekunder, men er lavere enn disken. Disken vil ha 5 ms, og vi ser at avstanden er relativt stort. Tallene til høyre for bildene er ment til å beskrive hvor lang tid det tar å aksessere minnet fra et virkelighetsperspektiv. I registerne så forestill at CPU´en er der vi sitter, det er datamaskinen og pulten vår, utrolig liten avstand altså. Cache befinner seg i bokhylla over deg, altså ganske liten avstand mellom CPU´en og Cache. Deretter har vi primærminnet som er avstanden mellom der du sitter og tiden det tar for å hente en bok i 2 etasje. Tillsutt har vi disk hvor dette vil tilsvare tiden det tar å reise herifra til månen eller noe lignende. Vi ser altså at tallene vist til venstre, ikke nødvendigvis viser hvor ille det er å aksessere minnet fra de høyere nivåene minnet som disken med 5ms. Men sett fra et virkelighetsperspektiv, så er det veldig ille. Lærdommen er altså å bruke minnet effektivt og helst unngå å aksessere minnet fra disk f.eks.



Tapes eller den øverste i minnehierarki, tar utrolig langt tid som det står i bildet over.

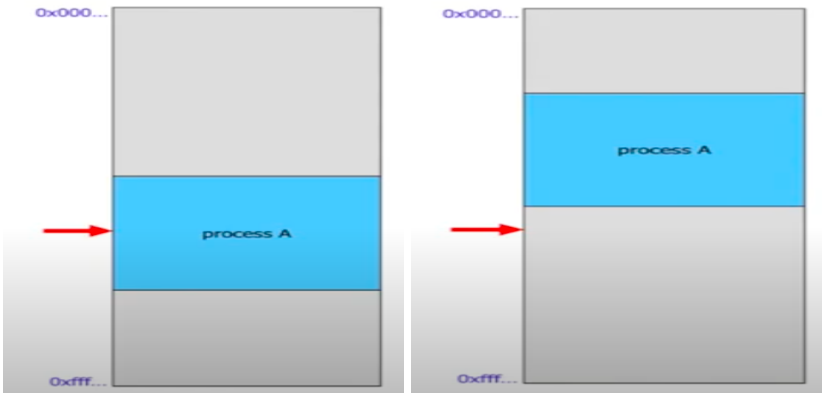
Adressering

Absolutt og relativt adressering

For å hente data fra minnet, har vi da adresser som fører oss til det gitte minneområde vi ønsker å aksessere. Vi skal se på to ulike måter for adressering i de neste avsnittene.

Absolutt adressering

Hardware bruker oftest absolutt adresser. I absolutt adressering så har vi da faste, reserverte minneregioner og lesing av data foregår ved å dekode byte-nummere i minnet. Dvs, vi kan ha at byte 1231421, betyr instruks 4 eller noe lignende. Når vi skal aksessere noe så har vi en bitstreng som referer til minneadressen direkte. Tilslutt så er det raskt å aksessere minne gjennom absolutt adressering. Dette er da for hardware, men hva med software?

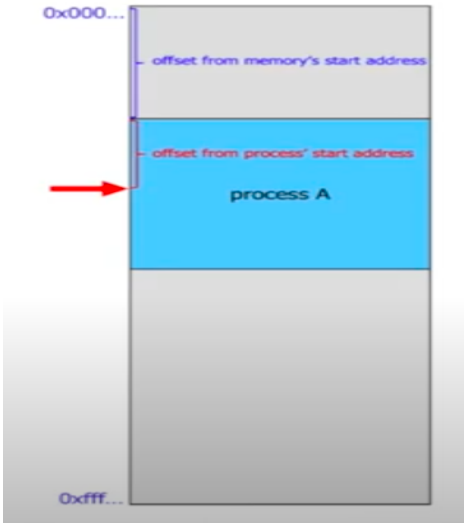


Kan software lese data gjennom absolutt adresser?

Svaret er nei, fordi si at vi har en prosess “A” som vist i bildet over som ligger i adresse (0x000fffff). Utfordringen vil være om en prosessen vil ligget i nøyaktig i dette adresse til enhver tid vi starter programmet. Gjennom bildet til høyre, ser vi at dette ikke er tilfellet hvor “A”, endrer posisjonen sin. Resultatet er avhengig av hvor prosessen ligger fysisk i minnet. Derfor kan vi ikke bruke absolutt adressering siden prosessen nødvendigvis ikke ligger i absolutt adresse (0x000ffff) til enhver tid. Hva blir da løsningen, det er relative adresser.

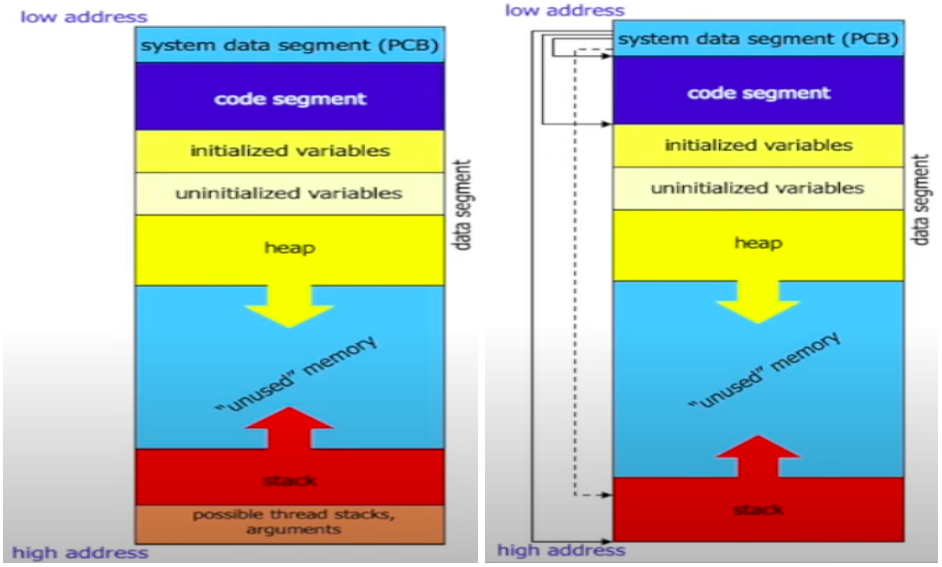
Relative addressing

De adressene som prosessen bruker, er stort sett de samme som prosessen internt bruker hver gang. Dermed, hvis man vet startadressen til en prosess så kan bruke relativ adressering internt i prosessen, slik at vi har en viss ide over hvor prosessen ligger rundt adresseområde. Så hvordan dette fungerer er at, man må vite startadressen til prosessen slik at vi kan finne en form for base-lokasjon. Base-lokasjon vil si relativt hvor prosessen ligger.



Bildet over beskriver hvordan relativ adressering fungerer hvor vi da har en start adresse over og en relativ adresse. Vi finner absolutt adressen til en prosess gjennom start adressen og relativ adressen. Så nå som vi har sett på de to ulike adresseringene så har vi da funnet ut hvordan vi får løst utfordringen med prosesser. Men layouten for prosessen vil ikke være slik som bildet over til enhver tid. En prosess har variabler, heaps, instruksjonspekere, registeret osv. Altså prosessen "partisjonerer" sitt minne hvor vi har kodesegmenter, instruksjonspekere, heaps, variabler og andre partisjoner som blir delt opp innad i minne til prosessen. Dette skal vi se mer på i neste avsnitt.

Prosessens minne "Layout"/utseende



Tekst-segment/Code-segment

- Ligger i de lave adressene.
- Lesing fra programfilen, f.eks execve() eller fork()
- Vanligvis er dette read-only

- Kan bli delt

Et data-segment

- Initialiserte globale/statiske variabler (data)
- Uinitialiserte globale/statiske variabler (BSS)
- Heapen befinner seg i dette segmentet
    - Dynamiske minne, allokeres gjennom bruk av malloc. F.eks når jeg skriver malloc i C, så allokerer vi plass til heapen.
    - Vokser mot høyere adresser, altså mot slutten av uinitaliserte variabler

Et stack segment

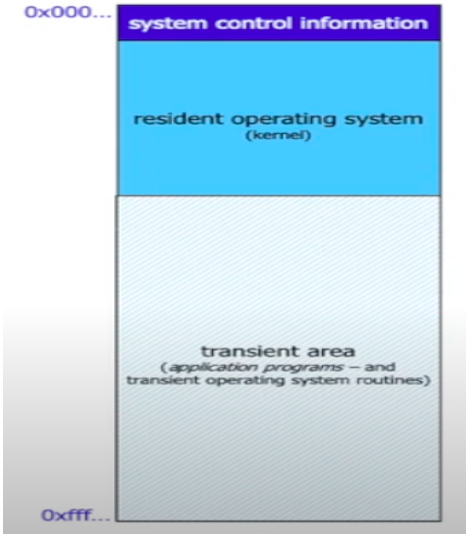
- Oppbevarer parameteret/variabler i en funksjon
- Oppbevarer register tilstander, f.eks kaller funksjonens “EIP”(Se første forelesning for disse tilstandene)
- Vokser mot lavere adresser som vi ser i bildet. Grunnen til dette er for å effektivt utnytte minneområdet. Så når både data og stack vokser mot hverandre, så utnytter vi minneområdet på en god måte. Dette ser vi i bildet over hvor vi har “unused memory”, men siden “stack” og “data” vokser mot hverandre, så vil vi bruke opp mye av den ubrukte minneområde i bildet.

System data segment/Process control block (PCB)

- Vi har ulike segment-pekere som vist i bildet over(den til høyre), hvor vi kan se piler som peker på starten av de ulike blokkene. Dette er da segment-pekerne som PCB har oversikt over. Årsaken er å holde oversikt over start-adressene til de ulike blokkene over som data,stack og kode segmentene.
- PID
- Program og stack pekere

På de aller høyeste adressene så har vi sikkert flere stacks for tråder, “command line” argumenter og enviroment variabler.

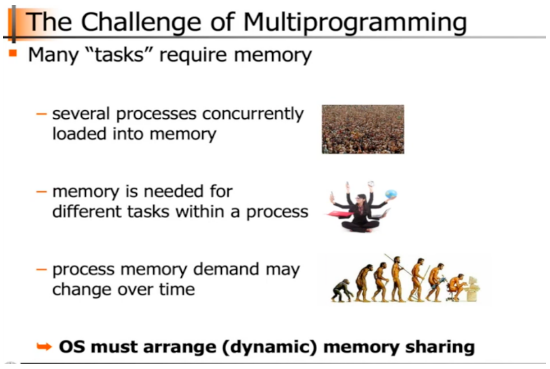
Globalt Memory Layout



Minne er oftest delt inni regioner som vi ser i bildet over

- I de laveste adressene så oppbevares operativsystemet
    - Som i bildet over så er “system control information” og “resident operating system” i denne delen.
- Den gjenværende området settes av til vanlige prosesser fra OS´et som kan byttes ut og inn. Som det også står i forelesningssliden “The remaining area is used for transient operating system routines and application programs”. Hvordan kan vi dele opp dette området, siden vi vet at prosesser blir oppbevart her, men hvordan skal vi dele opp “X” antall prosesser som kan ha “Y” ulike tilstander. Det skal vi se på i neste avsnitt.

Minnehåndtering for multiprogrammering(Mange prosesser, hvordan skal disse håndteres i minne?)



Bildet over, viser de ulike utfordringene vi har med multiprogrammering. F.eks hvordan skal prosessene fordeles i minnet, eller at minne er nødvendig for ulike arbeid innad i en prosess. Tilslutt at prosessets vil sikker kreve mer minne enn vanlig over tid. Dette er problemene som multiprogrammering står ovenfor, og løsningen er da det **siste punktet i sliden**.

Bruk av diske/sekundær lagring

Det å oppbevare alle programmene og deres data i minne, er sikkert umulig. Derfor må man ha en “trade-off”, en pågående veksling mellom primærminnet og sekundærminnet hvor vi da oppbevarer de relevante, viktige delene i primærminnet imens det som er ferdig eller som trengs til fremtiden forblir på disken.

Swapping

Går ut på at vi f.eks laster inn en prosess og lar den kjøre. Når den var ferdig og ble terminert så blir den tatt ut fra minnet og flytter den over til sekundær-medium som (disk, RAM, tape) med dets tilstand og data. Se på det som context-switching på en måte for å forstå dette bedre. Men dette blir ikke en god løsning som vi så lenger oppe med minnehierarki hvor å aksessere data fra sekundærminnet kan ta lang tid. Dette er en gammel måte som ble brukt i de tidlige dager, sammen med “Overlays”.

Overlay

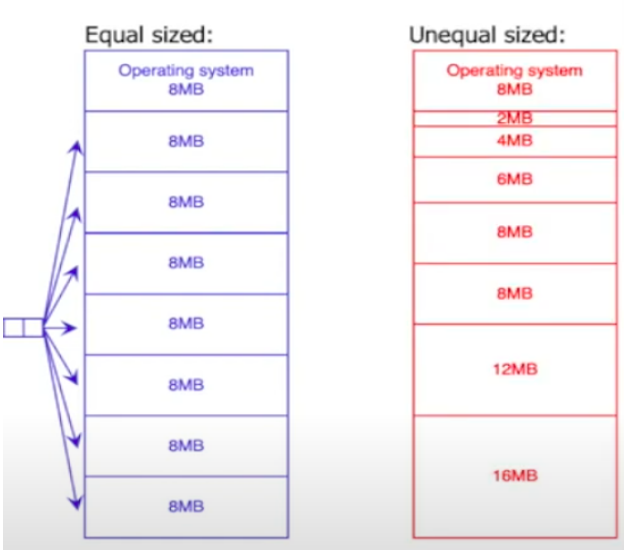
Handler om at utviklerne står fri til å manuelt styre programmet selv. Utviklerne har kunnskap til enhver tid for hva som er relevant for prosesseringen som skjer foreløpig. Primærminnet blir delt inn i “Overlays” eller segmenter som ble benyttet når programmet hadde nådd et punkt hvor disse “Overlaysene” ble nyttige. Dette er ikke en optimal løsning og derfor kommer “segmentation/paging” inn.

Segmentation/paging

Lignende Overlays, men hvor OS ser på systemet som helhet og ser hva som trengs internt for hver prosess. Ikke slik at utviklerne gjør dette, det er OS som har kontroll på dette. Men vi har fremdeles segmentene/pages som OS plukker til å velge de som er mest optimal for systemet i helhet og internt i prosessen. Hvordan utføres slik partisjonering, dette skal vi se i neste avsnitt hvor vi ser mer på ulike partisjoneringer.

Fixed partisjoner

Går ut på at man deler inn minnet i statiske partisjoner som har en størrelse bestemt under initalisering av systemet(bootstrap eller tidligere). Det er altså ikke fleksibelt, alt er gjort på forhånd og forblir slikt. Fordelene ved dette, er at det er enkelt å implementere siden man slipper å tenke på fleksiblitet. I tillegg til at det støtter de tidlige versjonene av “swapping” av prosesser.



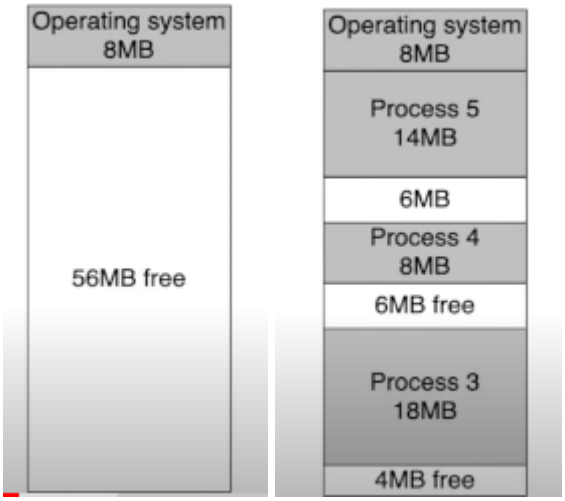
- Equal-partisjoner
    - Partisjonering hvor vi fordeler antall minne likt mellom minne. Vi ser til venstre i bildet at det er kun prosesser som krever 8mb som blir lagt inn her. Konsekvensen ved dette er ganske åpenbart som er da, hva om vi har store programmer med 126mb. I dette tilfellet så vil ikke disse programmene bli eksekvert siden deres minne er større enn det som er blitt tillatt. Ellers så må vi da hente deler av disse programmene i disken, noe som ikke burde gjøres i det hele tatt med tanke på utrolig treg aksesstid. Det kan også hende at noen programmer er ganske små som “8 kb”, som fører til at de ikke bruker hele partisjonen så man bortkaster minne.

- Unequal-size partitions
    - Partisjonering hvor det er ulikt fordelt, f.eks en vi kan ha en kø for 8 mb, en for 6 mb osv. Man får altså ulike prosesser som krever ulik minne. Men problemet som kan oppstå, er om vi har prosesser som bare er store eller små som fører til at minnet som ble satt av, ikke blir brukt eller ikke brukt opp mest mulig. Altså at det prosesser med store programmer ikke får mulighetene til å kjøre siden minneområdene ikke tilfredsstiller prosessenes krav. Eller at vi har små prosesser som ikke utnytter minnet som er blitt partisjoner effektivt.

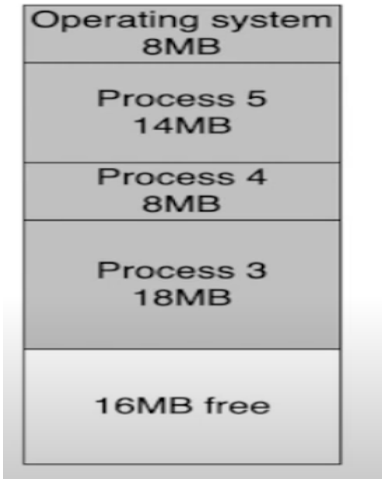
Dynamisk partisjoner

Dette er da ulikt fra “Fixed-partisjonering” hvor vi nå kan være fleksible med tanke prosesser som kommer inn i minne og at partisjoner kan bli lagd dynamisk i henhold til prosessenes

minne. Denne type partisjonering fungerer ved at minnet blir delt ved “run-time” hvor partisjoner blir lagd dynamisk.

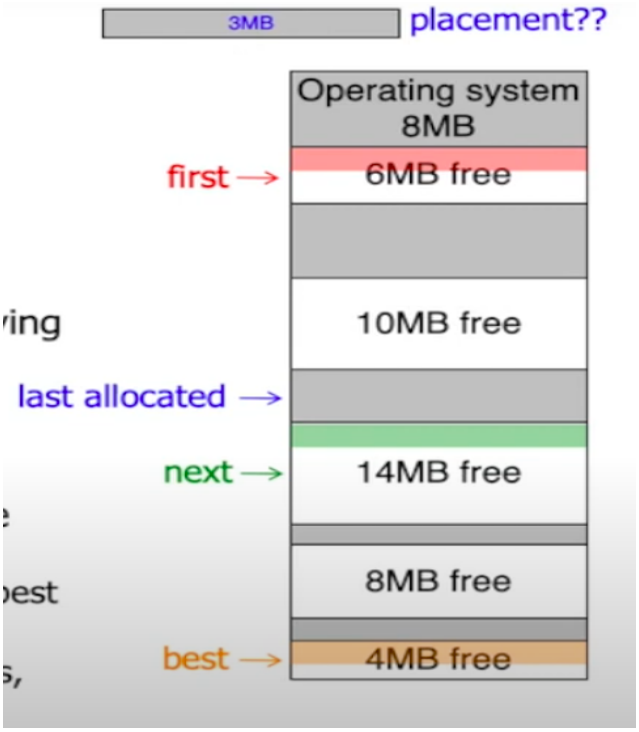


Som vist i bildet over så har vi i bildet til venstre, lagt at det er satt av 56 mb. I bildet til høyre så har derimot prosessene kommet inn i minnet og fått plass. Når prosessene er ferdig eller blitt terminert, så blir de tatt ut av minnet hvor vi ser tilfeller hvor minnet er fritt(f.eks 6MB free).



Utfordringen er at man kan få fragmentering, hvor vi kan få instanser som vist i bildet til høyre hvor vi har ledig plass i minnet (6MB, 6MB free, 4MB free), men en 8mb prosessor vil ikke få muligheten til å bli satt av i minnet. Problemet er at det ikke er nok plass til dette siden det ikke er noen partisjoner med 8mb. Denne utfordringen pleide man å først løse gjennom **“Compaction”.** Man tar de prosessene som kjører til enhver tid, og flytter disse til en del av minnet, og flytter alt andre som er ledig mot slutten. Som vi ser i bildet over, så er det slikt compaction funker. Problemet er at man bruker en del CPU-sykler på å flytte om de ulike prosessene til et sted, det som er blitt brukt av minne til et annet sted som bruker en del ressurser. Så man har en “trade-off” hvor fordeling blir slikt i bildet over, men konsekvensen blir tidsbruk på grunn av mange CPU-sykler og ressursbruk av prosesser.

En mulig løsning for å redusere bruken for compaction, er ved å ha en algoritme som velger de fornuftigste plassene å fordele prosessene på.



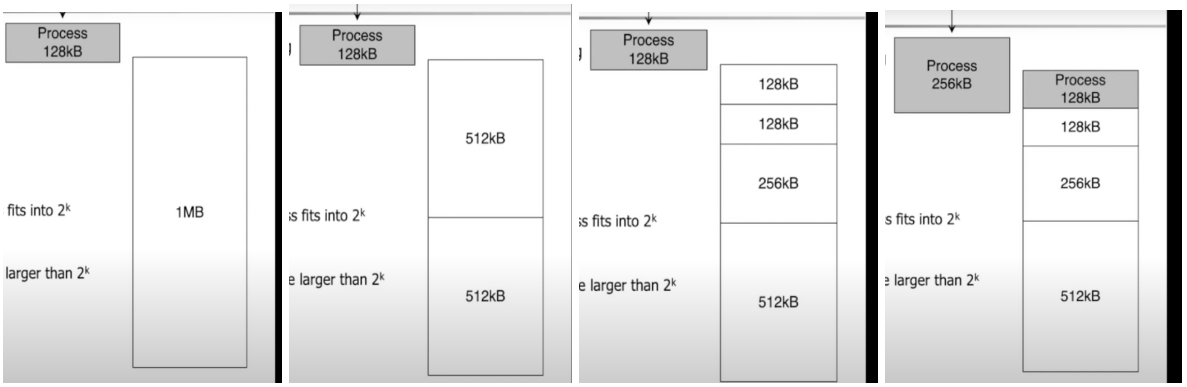
- **First fit**
  - Ved first fit, så leter man fra starten og finner den beste mulige plassen å legge prosessen på. Dette er den enkleste, raskeste og oftest den beste av de tre måtene som vi ser i bildet over(first, next, best)
- **Next fit**
  - Den leter fra da man sist allokerte et segment og som first fit, finner da den beste mulige plassen/partisjon.
- **Best fit**
  - Denne er komplisert siden den leter gjennom hele lista av partisjoner som er ledig og finner den som er best. Problemet er at denne er tregest kan føre til fragmenter som blir utrolig smale og er oftest den værste. Dette ser vi i bildet over siden om vi setter av 3mb i best-fit, så vil da få 1mb partisjon ledig og det kan hende at ingen prosesser har tilnærmet lik eller lavere mb enn 1mb. Dermed har vi bortkastet en partisjonering som aldri vil bli brukt på grunn av "best-fit".

Buddy System

Ideen går ut på at man deler opp en partisjonen, og lager 2 naboer som har lik partisjon. Hver partisjon har en størrelse som tilsvarer 2^k hvor 'k' er (L <= k <= U) . Kan sees på som

mergesort i IN2010 hvor vi deler opp en array i halvparten, også rekurseres dette og deler halvparten på hver sin array osv. Slik går “buddy-system” partisjoneringen ut på.

Når man skal tilordne minne til en prosess så finner man den minste ‘k’en slik at prosessen passer inn i 2^k. Eller en plass som er av størrelse 2^k. Hvis vi ikke har en ledig plass tilgjengelig, så splitter man opp den minste plassen som er større enn 2^k rekursivt. Dette kan være dårlig forklart og videoen gir en god animasjon over hvordan dette fungerer. Se bildene under for bedre forklaring:



Tilslutt så “merges” de ulike partisjonene med hverandre. Dette gjøres når prosessene er ferdig med å kjøre eller er blitt terminert. Men problemet ved dette systemet er om vi har prosesser som har minne som ikke tilsvarer nøyaktig 2-er potens. F.eks om vi har prosess med minne 513 kb. Dette er da en utfordring ved buddy-systemet.