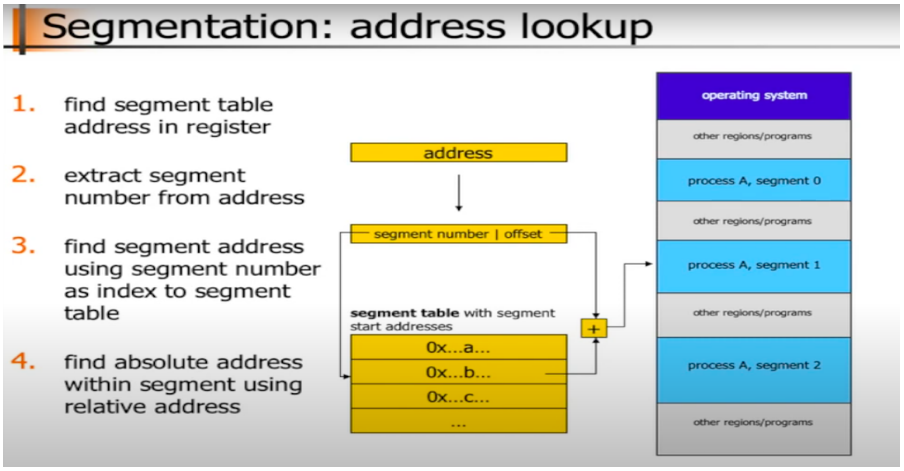
Segmentation

I første forelesning om minnet så har vi observert ulike utfordringer med å effektivt utnytte minnet. Vi har sett at partisjonering var en løsning hvor vi hadde “Fixed” eller “Dynamisk” hvor begge disse gav utfordringer. Men hva hvis vi bruker “Overlays” og ha en metode som kan hoppe dynamisk fra en partisjon fra den ene til den andre effektivt. Det er ideen bak segmentering hvor vi har en effektiv og dynamisk måte å utnytte minne på.

Segmentering/paging gir oss en løsning hvor vi har ulike segmenter som har forskjellige lengder som er bestemt av utvikler eller OS. Disse segmentene blir delt opp i minnerammer. Det er utvikleren eller kompilatorens oppgave å håndtere programmet i de forskjellige delene, altså at man flytter kontroll fra et segment til et annet. Viktig å nevne at en prosess vil bli ført inn i ulike segmenter som består av code-, data-, PCB-segmenten, så når vi flytter kontrollen, så vil det si at vi flytter fra et segment i prosessen, til en annen(Code-segment til data-segment). Man må være oppmerksom på at de ulike begrensningene til størrelsene for segmentene. Dette er for at det skal bli en effektiv måte å hoppe fra den ene segmentet til en annen.

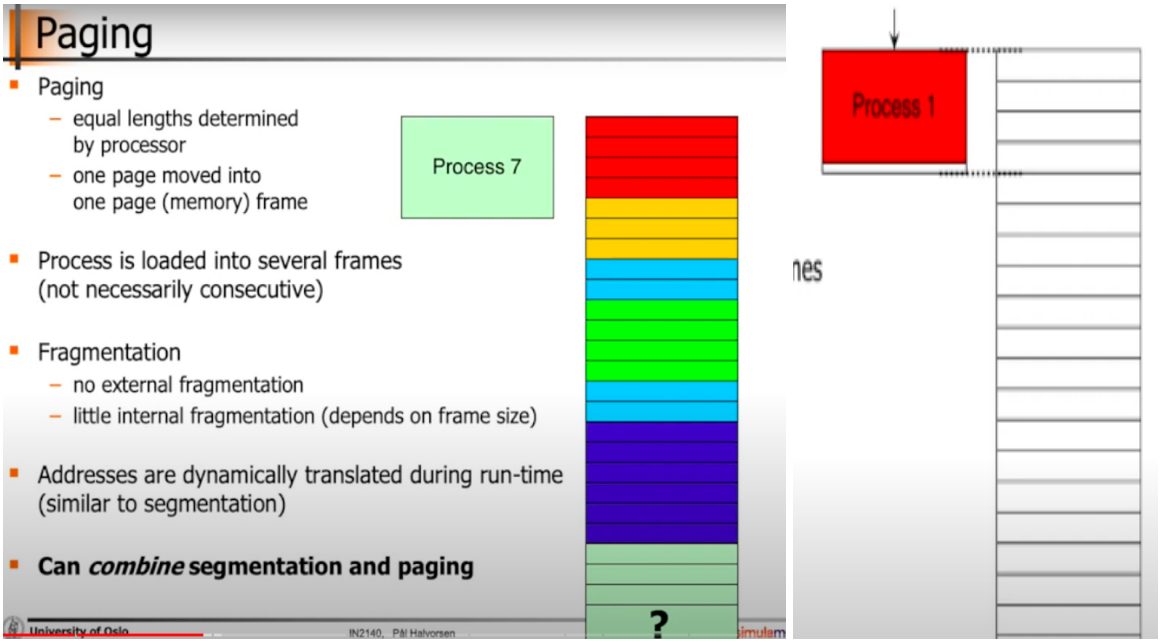
Fordeler ved segmentering, er at vi forholder oss til dynamisk partisjonering som vil si at prosessene kan ha forskjellig størrelse og segmentet vil forholde seg til dette. Det vil ikke forekomme noen interne fragmenter altså tilfeller hvor vi ikke har utnyttet en gitt “partisjon” effektivt i forhold til dets størrelse(8mb prosess, 7mb partisjon, ser at dette ikke blir effektivt). I tillegg til at det blir færre eksterne fragmenter. Det vil da si at det er færre tilfeller hvor minnet ikke blir fullt utnyttet og heller at minnet blir utnyttet på en fornuftig måte. Ulempen er at vi legger til ett steg til adresseoppslagget. Dvs, ikke bare må vi ha en baseadresse til prosessen og en intern relativ adresse for å finne prosessen i minne, nå har vi enda et steg til å vite hvilken partisjon som partisjonen befinner seg i.



Bildet over beskriver hvordan vi finner segmentet fra en gitt adresse. Så først må vi ha en "Segment-table" som har oversikt over alle start-adressene til hver segment. Vi husker ved relativ adressering at vi finner en gitt prosess ved å kombinere både startadressen og den relative-adressen til en prosess til å finne den interne base-adressen dets. Først må vi finne ut hvilken segment vi har å jobbe med, dette gjøres gjennom et segment-nummer. Nummeret er de øverste bitene i start-adressen til segmentet. Segment-nummer er indeksen til en gitt segment som beskriver "base-adressen" til det fysiske segmentet som vi ønsker å hente. Så ved å kombinere både nummeret, start-adressen og base-adressen så vil den faktiske, fysiske adressen til segmentet bli synlig.

Fordelen ved dette er at prosessene kan ha ulik størrelse, legge dem forskjellige steder i minnet osv, siden adresseoversettingen gjør det enkelt å finne prosessen vi skal finne. Men da tillater man at segmentene kan ha ulik størrelse, så når en segment blir kjørt inn, så må man sørge for at, når segmentet når sin maksimumsgrense, så er den ferdig med å kjøre. Dermed ville det hjulpet om segmentet hadde hatt fast størrelse, og da kommer vi til "paging".

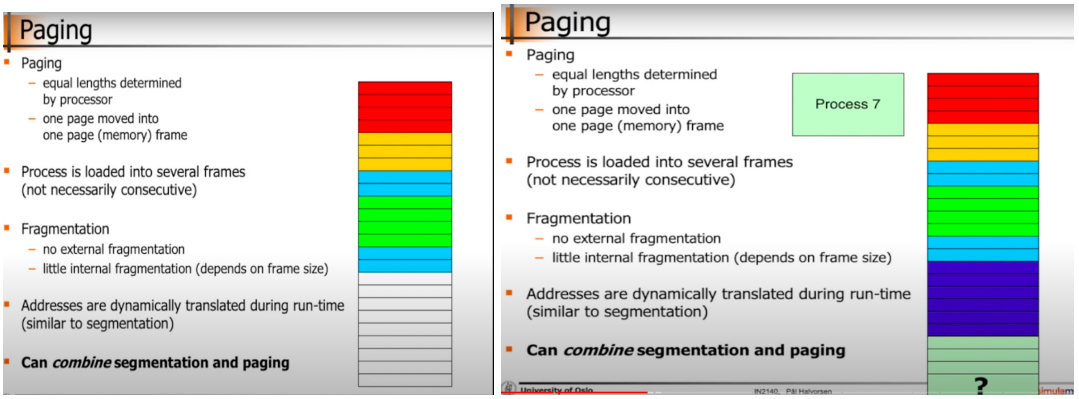
Paging



Paging går ut på at vi har det samme oppsettet som ved segment men forskjellen er at paging har faste størrelser som blir bestemt av en maskin. En page tilsvarer et gitt minneområdet. I en 32-bits arkitektur størrelsene i en page gitt i 4 kb, men man har muligheten til å gi 4 mb istedenfor som vi skal se senere.

Ideen er at man prøver å allokere så mange kontinuerlige “pages”, men hvis det ikke er mulig så settes de av til et annet sted. I bildet over, til høyre så ser vi et eksempel hvor “Prosess 1” skal settes av i minne. Vi ser at vi har plass til 4 pages, siden prosessen passer akkurat 4 pages. Resultatet er vist til bildet til venstre hvor vi ser helt øverst at de første pagesene er røde. Nå til et eksempel hvor man ikke har kontinuerlige pages som prosessen kan settes inn i. Hvis du ser mellom de lyseblå og grønne, så ser vi at de lyseblå ikke er kontinuerlige. Grunnen er at den lyseblå prosessen kom etter den grønne, altså grønn prosess tilsvarer “3”, imens lyseblå prosess tilsvarer “4”. Siden det ikke vi har kontinuerlig plass for den lyseblå, så ble prosessen delt opp hvor halve er over den grønne imens halve er etter den grønne.

Fordelen er at vi unngår fragmentering siden gjennomsnitt, det som blir kastet bort er halvparten av en page. Hvis vi ser på den røde prosessen altså “Prosess 1”, så var den litt under 4 pages. Men vi brukte opp 4 pages allikevel siden det var nok plass for det. Det blir altså litt internt fragmentering på grunn av dette tilfellet, men det er bedre sånn sett siden vi hindrer ekstern fragmentering og utnytter minnet meget bra. Den lille interne fragmentering missen er avhengig av størrelsen til hver page.

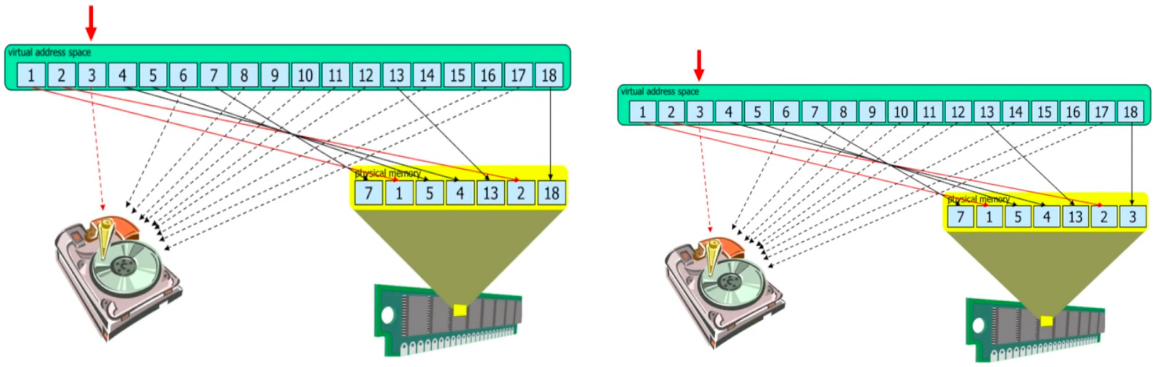


Problemet som kan oppstå, er om vi trenger mer minnet. Som du ser til bildet i venstre, så har vi “x” antall pages. Men til høyre så har vi plutselig en prosess som trenger flere pages enn det minnet har gitt oss. Hva skal vi gjøre da? Løsningen blir å innføre virtuelt minne.

Virtuell minne

Gjennom virituelt minne så får vi et virituelt lag på toppen av disken og primærminnet, som gjør at vi får et stort adresserom. Ideen er at forskjellige “pages” som prosessene blir plassert inni, blir lagt til dette adresserommet som kan peke videre til primærminnet eller disken(operasjoner som gjøres av sekundærminne/disken for å hente data fra primærminne, ikke at vi fysisk henter data fra sekundærminne). Virituelle adresser er oversatt til fysiske adresser gjennom en “page-table” hvor denne tabellen har pekere som peker til

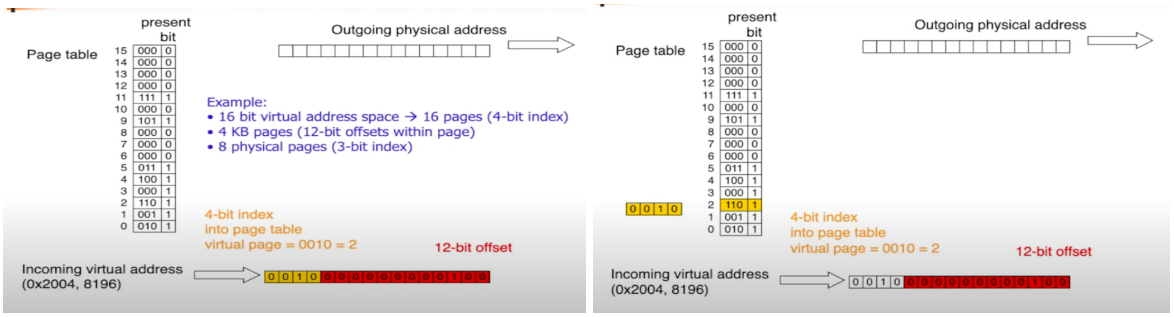
sekundærminne/primærminne. “MMU/minnehåndtereren”´en passer på at det vi forespør om, vil ligge i primærminne/sekundærminne.



Vi ser i bildet over, så har vi en del minneblokker i det virituelle minne som peker på disken og noen som peker på pagene i systemet. Hvis du ser pilen i bildet over, så er dette ment å vise et tilfelle hvor vi har en datablokk som primærminnet ikke hadde plass til og ble lagt inn i disken. Systemet har enten ikke lastet inn denne datablokken eller har tidligere flyttet denne datablokken tilbake til disken for å få plass til noe annet. Hva skal vi gjøre med minneblokk “3”, vel denne erstattes med page “18” i systemet hvor de bare bytter om plass (Se bildet til høyre).

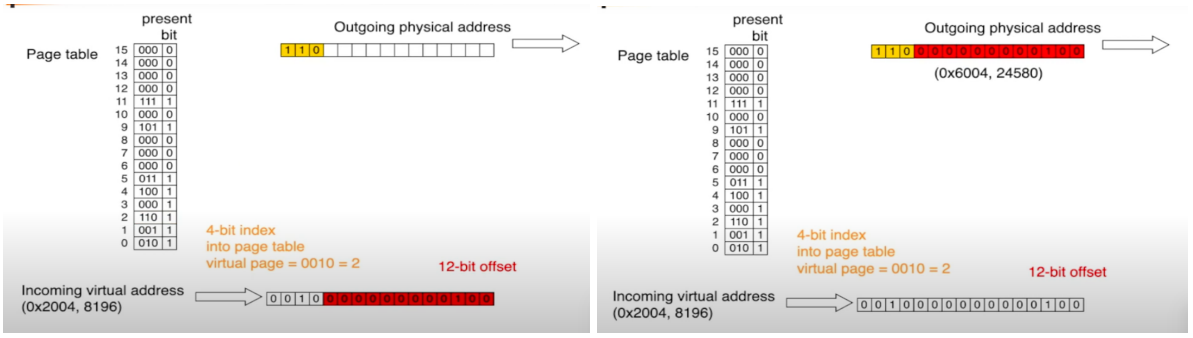
Bildene over beskriver enkelt hvordan dette er satt opp altså med at noen blokker i det virtuelle adresseområdet peker på disken og systemet. Men hvordan er det dette egentlig sjekkes opp, dette skal vi se nærmere på i neste avsnitt om “Memory Lookup”.

Memory Lookup

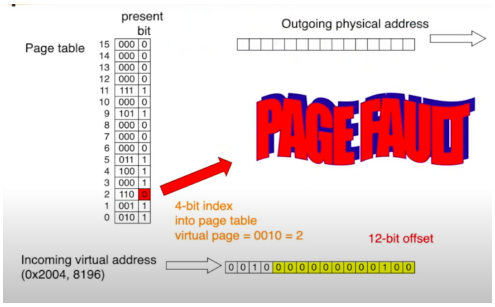


La oss fokusere på disse bildene for nå og casen som ble tatt opp. Casen er nevnt i eksempelet for bildet til vensteret. Vi husker ved segmentering at de hadde en indeks for som var altså de første bitene hvor disse kombinert med startadressen til et segment hjalp oss med å finne en base-lokasjonen til en prosess. Indeks fungerer likt i bildet over hvor vi da har en indeks som beskriver hvor i denne “page-tabelen” som den fysiske adressen faktisk ligger i. Så da sendes med den 4-bit indeksen mot de virituelle minne hvor da det

virituelle minne prøver å finne adressen som tilsvarer denne indeksen i page-tabellen. Vi har i page-tabellen en “present-bit”, dette bittet forteller oss om pagen finnes i minnet eller i disken, som vi så i det forrige bildet helt over.

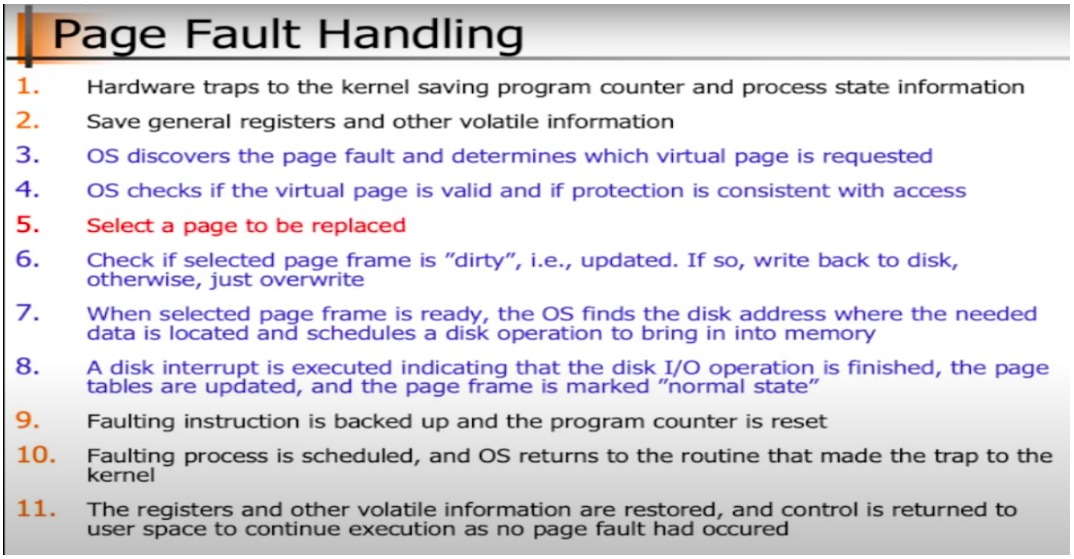


Når vi har lest inn den 4-bits indeksen og satt den inn i det fysiske minneområdet, inkludert den 12-bit offsetten, så vil vi da få ut en bitstreng som tilsvarer det fysiske minneområde til pagen.



Problemet som oppstår er at present-bittet for den samme indeksen brukt over, er blitt satt til 0. Vi fant den fysiske adressen som vist i forrige bildet så derfor ble present-bittet endret. Da er ikke den pagen i primærminnet, og dette håndteres hvor vi kaller dette page-fault.

Page-fault håndtering



Det som skjer, er at vi for en context-switch, hvor vi får en prosess nede i kjernen som får varsel om at det har oppstått en page-fault hvor vi lagrer tilstanden til prosessen som kjørte. Deretter prøver kjernen å finne ut hva det var som skjedde, hvor kjernen prøver å finne hvilken adresse med present-bit “0” var det som førte til page-fault.

Hvis vi husker de to bildene under “Virtuelt minne” så er det dette scenarioet som oppstår hvor da vi prøver å bytte ut prosessen i kjernen med en prosess i systemet til å bli offeret. Først når vi skal bytte ut prosessen i kjernen med prosessen i systemet, så må vi forsikre om at prosessen i systemet har blitt oppdatert for å ivareta all data. OS´et finner prosessen som blir offeret og skedulerer en diskoperasjon som har ansvar om å sette prosessen inn i minnet. Når disken er ferdig, får man et interrupt hvor page-tabellen blir oppdatert hvor man markerer pagen til å settes til “normal state”.

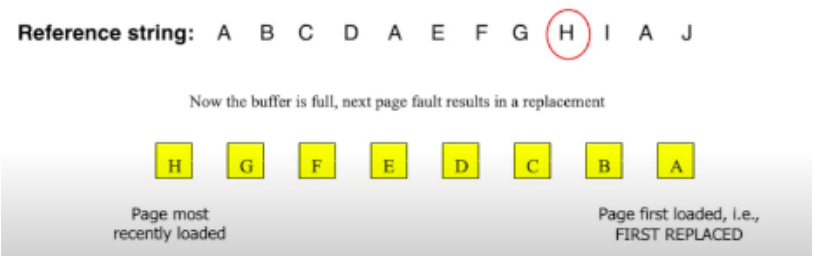
Tilslutt vil man gjøre hele prosessen som vi skrev under i “Memory Look” hvor vi henter indeksen, sjekker indeksen i page-tabellen, forsikrer om at present-bittet er 1, så finner vi det fysiske minneadresset til prosessen gjennom offset + indeksen til adressen. Present-bittet vil bli satt til 1 siden vi utførte stegene over. Problemet vi står ovenfor nå, er å velge ut den prosessen som skal bli offeret for håndtering av page-fault. Hvilken “page” skal OS´et velge ut, dette skal vi se nærmere på i neste avsnitt.

Page Replacement Algorithms

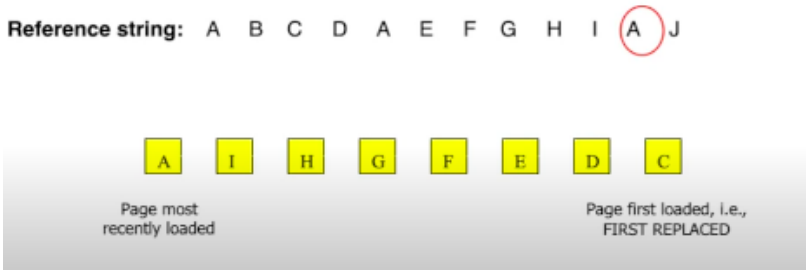
Vi har algoritmer som velger ut en gitt page som blir offeret ved håndtering av page-fault. Disse algoritmene skal vi se nærmere på under.

FIFO

- All pages in memory are maintained in a list sorted by age
- FIFO replaces the oldest page, i.e., the first in the list



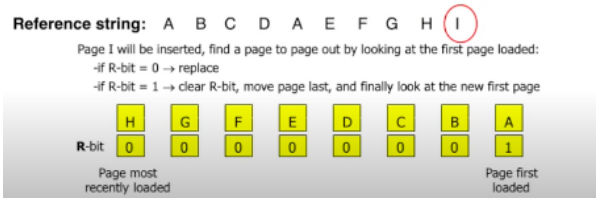
Her ser vi hvordan FIFO håndterer dette hvor bufferet(Gule boksene i bildet) er fylt inn og vi har ikke plass til prosess “I”. Det som skjer er at prosess “A” blir den første til å komme ut og kastes inn i disken, og “I” settes helt på starten.

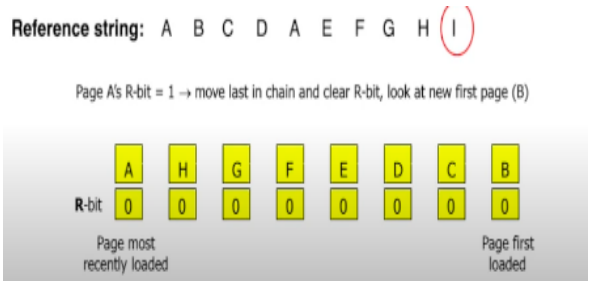


Problemet her blir at “A” nettopp kommer tilbake etter at vi kastet den i disken. Da må vi kaste ut den som er sist i køen og sette “A” helt i begynnelsen. Fordelene er at det er enkel å implementere, skaper lav overhead. Men ulempene er at disk-aksessering er dyrt og tilfellet hvor “A” ble kastet ut så tatt inn igjen, er dyrt også.

Second Chance

- Modification of FIFO
- ***R*** bit: when a page is referenced again, the R bit is set, and the page will be treated as a newly loaded page

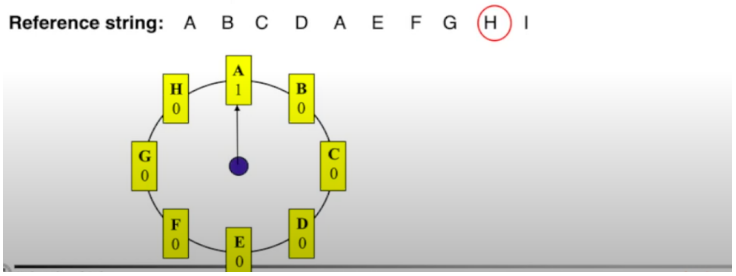




Denne algoritmen er en modifisering av FIFO, men hvis det er tilfeller hvor en prosess blir gjentatt igjen, så vil den få en “Second chance”. I bildet så har vi at prosess “A” gjentas to ganger. Denne vil få en referansebit og blir til “1”. Når den kommer sist i køen som i bildet over, så skulle den egentlig ha blitt kastet ut og prosess “I” skulle ha kommet først. Men siden den har referansebit “1”, så vil “A” bli satt helt på starten og “I” blir da lagt foran “A”. Fordelen er at vi håndterer problemet ved FIFO, men problemet er at listeoperasjonene som dette med å bytte om “A” fra start til slutt skaper stor overhead. Så derfor blir “Clock” en bedre algoritme alternativ.

Clock

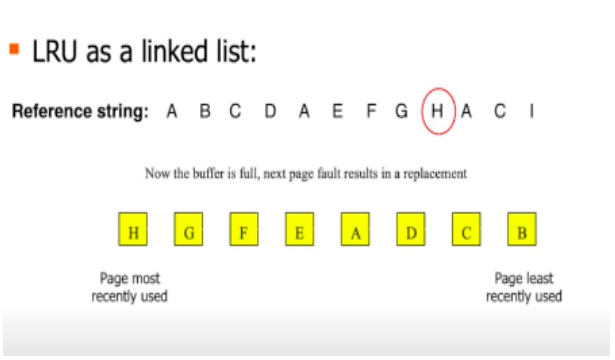
- More efficient implemention ***Second Chance***
- Circular list in form of a clock
- Pointer to the oldest page:
  - R-bit = 0 → replace and advance pointer
  - R-bit = 1 → set R-bit to 0, advance pointer until R-bit = 0, replace and advance pointer

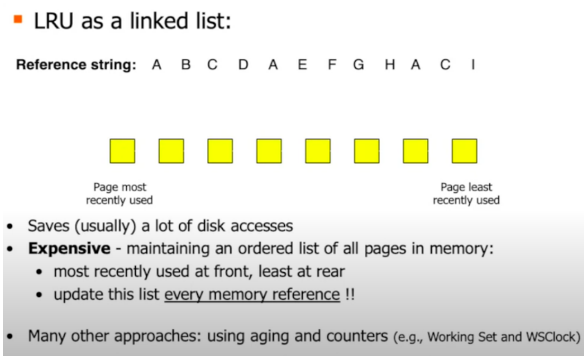


Denne algoritmen er et bedre alternativ hvor vi har et ring-buffer med en peker i midten som peker på de ulike pagene. Pekeren vil peke på den "pagen" som ble satt inn sist. Dette er et bedre alternativ enn "Second-chance" siden vi slipper å måtte bytte prosesser innad i listen som fører til at vi unngår overhead. I tillegg så fungerer "Second-chance" implementasjonen likt her siden "A" har referansebit 1, så kommer page "I" og da vil ikke "A" bli fjernet men "B" istedenfor.

Least Recently Used (LRU)

Selvom "Clock" var et bedre alternativ enn "FIFO" og "Second-chance" så er "LRU" et bedre alternativ. Årsaken er at den sparer mange disk-aksesser, noe som oppstår i de tre overnevnte algoritmene. Tanken bak algoritmen er at man erstatter den pagen som har lenge siden blitt sist referert. Det finnes ulike måter å lage denne algoritmen på, men her er eksempelet av en algoritme implementasjon under.





Vi har en lenkeliste hvor pagene blir satt inn i lenkelisten hvor de til venstre er de som ble nettopp satt inn, imens de til høyre er de som blir brukt sjeldent. Slikt minimeres antall diskaksesseringer siden de pagene som gjentar seg, de vil bli plassert lenger fremme i listen. Imens de som ikke har blitt brukt så ofte vil plasseres lenger mot høyre. Men denne

algoritmen har ulemper hvor problemet er at man kan ha en kostbar algoritme altså en algoritme implementasjon som er komplekst og kostbart. Som det å holde styr på algoritmene som ble sist referert, endre strukturen i lenkelisten hvis vi har noen pages som gjentar seg osv.