OS H2018

Innholdsfortegnelse

[Introduksjon 1](#_Toc531531577)

[Kjernemodus og brukermodus 3](#_Toc531531578)

[Prosesser 4](#_Toc531531579)

[Tråder 5](#_Toc531531580)

[Kjernen 5](#_Toc531531581)

[Minneadministrasjon 6](#_Toc531531582)

[Gammelt kompendium 6](#_Toc531531583)

[Allokering av minne 7](#_Toc531531584)

[Teknikker for minneadministrasjon 8](#_Toc531531585)

[Virtuelt minne 8](#_Toc531531586)

# Introduksjon

Et operativsystem er et program som administrerer en datamaskins ressurser for brukeren og for applikasjoner og programmer. Med operativsystem følger en del utfordringer. Dette er utfordringer som sikkerhet, portabilitet, pålitelighet, og responstid.

På samme måte som andre programmer er operativsystemet et program som kjører på PC-en din. Operativsystemet er dermed plasser mellom maskinvaren og brukerprogrammene. Et hvert program ønsker å gjøre forskjellige operasjoner, systemkall, som skriving og lesing. Dette må de spørre operativsystemet om å gjøre, og det er dens oppgave å dermed fordele ressursene rettferdig.

Maskinvare-komponenter kalles ofte ressurser i operativsystem sammenheng, og operativsystemets hensikt er å lage en standard modell av maskinvaren. Dette kalles ressursabstraksjon. Dette betyr at operativsystemet tilbyr standardiserte funksjoner for å for eksempel lese eller skrive til disk, slik at programmereren ikke trenger kunnskaper om hvordan disken fungerer.

Programmer utnytter ressurser som CPU og minne ved å kalle på operativsystemet i såkalte systemkall. Operativsystemet sørger for såkalt *ressursisolering* slik at programmer ikke får tilgang til de samme ressursene til samme tid og dermed kan lese fra samme dataområder. Noen ganger trenger vi likevel at programmer kommuniserer og da kan operativsystemet legge data i et felles minneområde der begge programmer har tilgang. Her sier vi at programmene *deler* på ressursene.

**Programmer og prosesser**

Kort oppsummert kan en si at en prosess er et program under utførelse. Når vi sier program tenker vi ofte på koden slik den ligger på en fil, «død», altså ikke kjørende, på en disk. Når programmet lastes inn i minnet og kjøres, kaller i det en prosess. Denne prosessen bruker programmets instrukser og maskinens ressurser for å gjøre det den skal. Når programmet lukkes, avsluttes prosessen, og det eneste vi sitter igjen med er programmet, og dataene det har generert.

Alle ressurser et operativsystem bruker, for eksempel en fil, har en *deskriptor*, altså en beskrivelse. Denne kan for en fil inneholde en dato filen ble opprettet, størrelse på filen, og adressen til datablokkene og tilgangsrettigheter. Denne deskriptoren er meget viktig for operativsystemet når administrere ressurser.

**Prosesser**

Som sagt er en prosess et program under utførelse. Vi sier at en prosess har følgende komponenter:

* Programkoden. Instruksjonene som bestemmer hva programmet skal gjøre.
* Dataene som programkoden bruker under kjøring. Dette vil alltid være referanser til dataområder.
* Ressurser som kreves av programmet, for eksempel en fil som må åpnes for å hente data.
* Status. Med denne mener vi om programmet venter på ressurser, om det venter på CPU ressursen, eller om det kjører på CPU ressursen.

**DMA**Direct Memory Access er at IO-kan kommunisere direkte med minnet uten å gå gjennom CPU.

**Avbrudd**

Avbrudd gjør at CPUen kan utnyttes mye mer effektivt. Ved for eksempel skriving til en enhet som er mye langsommere enn CPU, kan avbrudd brukes til å fortelle CPU at enheten er klar til å fortsette skrivingen, i stedet for at CPU må «vente» på at enheten er klar.

Når et avbrudd kommer til CPU vil CPU gjøre ferdig den instruksen den holder på med, og deretter utføre de instruksene avbruddet sier at den skal utføre. Vi sier at CPU begynner å utføre et avbruddsprogram. Hva dette programmet gjør avhenger av hvilket avbruddssignal CPU har mottatt.

Vi kan dermed si at multi-tasking teknikken til operativsystemet (teknikken som gjør at nye prosesser kan bruke CPU dersom en annen prosess trenger å bruke IO) avhenger av en god avbruddsmekanisme.

**Tildeling av ressurser og kjøretid på CPU**

Når operativsystemet skal tildele ressurser, tildeler den disse til hele prosesser. Dette vil si at alle tråder i en prosess deler på ressurser. Når det kommer til kjøring på CPU, tildeler operativsystemet dette til tråder. Det vil si at enkelte tråder kjører på CPU individuelt.

Prosesser trenger ikke spesifikt å be om å kjøre på CPU, i med at dette er hensikten til hele prosessen. Kjøring på CPU tildeles derfor automatisk gitt at en prosess har alle de ressursene den trenger. Dersom en prosess derimot trenger ressurser fra for eksempel IO, må prosessen gjøre et systemkall og be om å få ressurser. Operativsystemet bruker prosessens deskriptor når den skal flytte prosesser mellom CPU-kø og diverse vente-køer.

**Minneadministrasjon**

For at CPU skal kunne kjøre flere prosesser samtidig må operativsystemet kunne administrere minnet slik at ingen prosesser får tilgang til hverandres minnelokasjoner, slik at flere prosesser får plass i minnet og slik at de kan dele et område i minne dersom programmene skal utveksle data. I moderne operativsystemer benytter man virtuelt minne. Dette fungerer ved at operativsystemet tillater prosessen å se et mye større minne enn det som er tilgjengelig i primærminnet. Den gjør dette ved å ta i bruk harddisken. Vi kan si at virtuelt minne er satt sammen av RAM og harddisk.

Denne minneadministrasjonen kan foregå på forskjellige vis. Vi har *sidedelt minneadministrasjon* og *segmentdelt minneadministrasjon*. I moderne operativsystemer benytter vi en kombinasjon av disse som kalles *minneadministrasjon med sidedelte segmenter*. Funksjonen til det virtuelle minnet er å la flere prosesser ligge i minnet samtidig og dermed gjøre det enklere for CPUen å kjøre flere av dem samtidig.

## Kjernemodus og brukermodus

Tidligere hadde prosesser tilgang til hele minnet. I nyere operativsystemer reserveres mindre deler av minnet til hver prosess. Vi sier at en prosess kan befinne seg i to forskjellige moduser.

I brukermodus kan en prosess utføre instruksjoner mot sine egne dataområder i minnet. Men så snart en prosess trenger å lese fra tastatur eller skrive til skjerm, må den utføre systemkall. Da sier vi at prosessen går over i kjernemodus. Denne modusen er fullstendig styrt av operativsystemet. På denne måten kan operativsystemet sikre at prosessen ikke gjør ulovlige operasjoner. Eksempler på slike operasjoner kan være å lese fra andre prosessers minneområder eller skrive over viktige systemområder på disken.

**I brukermodus** er det brukerprosessens egen instruksjoner som opererer mot prosessens egne data.

**I kjernemodus** er det operativsystemets egne instrukser som utføres mot operativsystemets egne dataområder.

**Driverprogrammer**

Dette er programmer som kommuniserer med IO-kontrollere. De er altså helt vanlige programmer som i stedet for å operere med data i minnet, opererer med registrene til IO-kontrollere. Siden hver prosess ikke har noe med ressursadministrasjonen å gjøre vil en *write* funksjon i en prosess virke som en direkte kobling til den enheten det skal skrives til. I realiteten er det operativsystemet som kjører funksjonen write opp mot den aktuelle IO enheten.

# Prosesser

Som vi har sett er prosesser programmer under utførelse. Flere prosesser kan kjøre samme program, for eksempel forskjellige steder i programmet. Når man skifter mellom prosesser må operativsystemet huske hvor langt prosessen var kommet slik at den kan fortsette her ved en senere anledning. Dette er spesielt viktig ved for eksempel avbrudd. Da må prosessen som kjørte før avbruddet kunne fortsette der den slapp når avbruddet er ferdig.

Prosesser kan kjøre «samtidig» fordi operativsystemet sørger for å utføre context-switches slik at alle prosessene kjører litt og litt, og dermed fremstår det som de kjører samtidig.

Et operativsystem kan suspendere en prosess dersom den bruker for mye tid. I et slikt tilfelle må det lagres en del informasjon om prosessen slik at det blir lett å gjenoppta arbeidet den gjorde. Slik informasjon lagres i en såkalt *prosessdeskriptor*, kalles også PCB – Process Control Block. Når denne prosessen får lov til å kjøre igjen vil OS gjenopprette status igjen til slik den var da prosessen avsluttet.

Det er også viktig at forskjellige prosesser har sine egne dataområder dersom de ikke skal kommunisere. Dette gjelder også dersom de to prosessene kjører samme program, det er likevel ikke sikkert vi ønsker at disse skal ha tilgang til samme data, for eksempel hvis programmet kjøres av to forskjellige brukere.

Vi skiller mellom cpu-kø og IO-kø. I cpu-køen ligger alle de prosessene som har fått alle ressurser de trenger. I IO-køen ligger de prosessene som fortsatt venter på ressurser, for eksempel data fra disken, eller inndata fra tastaturet.

**Ressurser**

På samme måte som prosesser har en deksriptor, har også ressurser en deskriptor, en såkalt ressursdeskriptor. Når en ressurs ikke er opptatt, vil dens deskriptor havne på en liste over ledige ressurser.

**Oppstart av et operativsystem**

Det første som skjer er at enkelt enkle programlinjer i ROM sørger for at et enkelt boot-program hentes fra disken og inn i minnet. Dette programmet kan igjen hente operativsystemet inn fra disken og laste det inn i minnet. Dette er riktignok en primitiv versjon av operativsystemet siden det ikke finnes noen prosesser der. Den første oppgaven til OS blir derfor å sette opp et miljø for prosesser og ressurser. Tidlig i oppstarten av maskinen oprettes også en init-prosess. Dette er den første prosessen som opprettes på maskinen. Deretter lages en funksjon som kan opprette andre prosesser. Dette er *fork* funksjonen, og denne benyttes for å lage flere prosesser av init-prosessen, prosessers mor. Det lages flere kopier av denne prosessen, men de har alle forskjellige pid nummer(processID).

**Pre-emptive prosesser**

Pre emptive prosesser er prosesser som operativsystemet har full kontroll. Moderne operativsystemer er gjerne pre-emptive. De har altså full kontroll over alle prosesser. Det vil si at de kan bestemme hvor lenge en prosess skal bruke CPUen. Dersom prosessen overskrider denne tidsgrensen, vil operativsystemet fjerne prosessen fra CPUen, suspendere den. Den må da tilbake i kø.

**Non-preemtive (Cooperative multitasking)**

Dette er motsatsen til pre-emptive operativsystemer. Her samarbeidet operativsystemet med prosessene og en prosess kunne selv gi fra seg CPU til en annen prosess. Ulemper med dette er at prosesser ikke nødvendigvis gir fra seg CPU og dermed skaper lang ventetid. Enda verre er det dersom prosessen «henger seg» før den får gitt fra seg CPU.

## Tråder

Vi har sett viktigheten av isolerte dataområder, altså at ingen prosesser har tilgang til andre dataområder enn sitt eget. Dette er av sikkerhetsmessige grunner. Vi så tidligere på at ressurser tildeles til prosesser. Innenfor prosesser kan det likevel eksistere tråder. Disse kan alle sammen tildeles CPU tid. Dette vil si at fler tråder kan kjøre på CPU samtidig, men forskjellige steder i programkoden til prosessen. Alle trådene har tilgang til prosessens ressurser. Det er altså ingen beskyttelse mellom tråder, på samme måte som det er beskyttelse mellom prosesser. Dette er heller ikke ønskelig da delte ressurser er grunnen til at vi ønsker å ha tråder. Trådene utveksler data mye enklere enn det prosessene gjør fordi de ikke trenger å gå via kjernen da de deler minne. Trådene har sin egen instruksjonspeker, og på denne måten kjører de forskjellige steder i programkoden samtidig.

Det er ønskelig å bruke tråder dersom man ønsker å kjøre flere aktiviteter samtidig, og at disse aktivitetene har tilgang til det samme adresseområdet.

Ulemper med tråder er at hele prosessen krasjer dersom en tråd krasjer.

**Viktige funksjoner**

Thread\_create – lager en ny tråd, og lagrer informasjon om den i thread

Thread\_yield – Programmereren legger inn denne funksjonen for å få en tråd til å gi fra seg CPU til en annen tråd. Schedulern kan når som helst velge å gi CPU tilbake til den tråden som opprinnelig kjørte

Thread\_join – venter på en gitt tråd med å fullføre, før join function er ferdig.

Thread\_exit – finish the current thread

# Kjernen

Som vi har sett på holder operativsystemet på med mye viktig administrasjon. Dette innebærer administrasjon av IO utstyr, altså ressurser, administrering av CPU køen, også en ressurs, men som innebærer å bestemme hvem som skal få bruke CPU. Videre er minneadministrasjon en viktig oppgave operativsystemet har.

**Systemkall**

Et systemkall er at et program ber om en tjeneste fra operativsystemet. Dette er nødvendig så snart et program ønsker å lese/skrive data utenfor sitt eget adresseområde. Grunnen til at man har systemkall er så programmer ikke skal kunne få adgang til maskinvare på operativsystemet. På denne måten gir systemkall en veldefinert og sikker tilgang til operativsystemets tjenester.

**Kjerne- versus brukermodus**

I brukermodus kan programmer bare aksessere egne ressurser, altså ressursene i eget adresseområde. Denne modusen er lavt privilegert, og kan dermed ikke aksessere maskinvare eller operativsystemets ressurser. I denne modusen har også alle prosesser egne adresseområder.

I kjernemodus er man høyt privilegert som betyr at man kan aksessere alle ressurser. En feil i denne modusen vil krasje hele operativsystemet.

Brukerprogrammer svitsjer hele tiden mellom disse modusene gjennom en rekke systemkall. Dette foregår ved at et program ber operativsystemet om å utføre en handling, for så å få returnert resultatet av handlingen.

Et systemkall er definert som at et program ber om en tjeneste fra kjernen. Vi skal nå se at det finnes flere typer kjerner:

* Monolitisk kjerne: Operativsystemets egne instruksjoner kjører i samme adresserom, i kjernemodus. Dette er altså et stort program som inneholder hele operativsystemet, inkludert driverne. Dette er enklere å implementere, men har den ulempen at en feil i for eksempel en driver, vil krasje hele systemet. Til gjengjeld er denne kjernen raskere da den har færre kodelinjer enn en mikrokjerne.
* Mikrokjerne: Her er operativsystemet delt opp i prosesser som kjører i vanlig brukermodus. Dette er mer modulært og dermed lettere å vedlikeholde. I en slik kjerne er altså flere av opsys-tjenestene flyttet ut av kjernen og inn i et sett av tjenere, som kommuniserer via en mikrokjerne. Her er da minst mulig i kjernemodus, og mest mulig i brukermodus. En slik kjerne er enklere å vedlikeholde, men et stort antall systemkall og kontekstsvitsjer kan senke hastigheten til systemet.
* Hybridkjerne. Denne kjernen ligner en mikrokjerne, men har mer programkode i kjernemodus for å øke hastigheten på systemet.

En mikrokjerne bruker mer minne enn en monolittisk kjerne, og er tregere siden kommunikasjonen går mellom tjenere og kjernen. Til gjengjeld er den enklere å vedlikeholde.

En hybridkjerne har noen flere tjenester i kjernemodus enn mikrokjernen, men kjører utstyrsdrivere i brukermodus. Det er en slik type kjerne som brukes i alle kommersielle operativsystemer. En slik kjerne har enkelt design og god kjørestabilitet.

Kontekst svitsj – når cpu bytter hvilken prosess som kjører

# Minneadministrasjon

## Gammelt kompendium

Minneadministrasjon handler om hvordan operativsystemet håndterer prosessene som kjører på en datamaskin og hvordan den får plass til disse på minnet (RAM). Det handler også om hvordan operativsystemet får plass til prosesser og deler av prosesser som ikke kjører akkurat nå på harddisken, og hvordan det bruker dette som en utvidelse av minnet. Administrasjonen av delt minne er også en viktig oppgave, samt å sørge for minst mulig aksesstid.

En viktig del av minneadministrasjonen er også å utsette adressebindingen til programdelene er lastet inn i minnet. På denne måten kan man plassere programmene på vilkårlige steder i minnet.

Når vi har flere programmer i minnet vil de være på forskjellige adresser hver gang de kjøres. Hvordan skal programmene da vite hvilke adresser de da skal bruke for å for eksempel hente data? En mulighet er å kompilere programmet på nytt hver gang det starter, men dette er meget tungvint da vi hver gang må kompilere programmet. En mye enklere måte er å bruke et *baseregister*. Dette registeret angir startadressen til programmet i minnet. Det kan altså si at et program begynner på adresse nr. 600 i stedet for 0.

Selve adressene som brukes i programmet sier vi er virtuelle. I eksempelet over med programmet som starter i adresse nr.600, vil for eksempel adressene ta utgangspunkt i 0 som startadresse, men siden programmet faktisk starter i adresse 600, må vi gjøre en omregning fra virtuelle adresser til fysiske adresser. For slike omregninger har vi en egen enhet som adressene går gjennom for omregning. Hver gang CPU setter en adresse ut på adressebussen skjer en slik omregning. Det er på denne måten vi kan legge programmer hvor som helst i minnet. Det eneste som må gjøres er å holde baseregisteret oppdatert. Baseregisteret er faktisk en del av prosessdeskriptoren til prosessen.

Dette med å kunne *relokere* programmer er meget viktig dersom flere programmer skal kjøres samtidig. Man kan da ikke regne med at de får samme adresser hver gang. Det er også meget viktig dersom man kjører programmer som til sammen er større enn minnet og dermed må ha noe data på sekundærlagring. I slike tilfeller må programmer ofte hentes inn og ut av minnet og man kan heller ikke der regne med at de får samme adresser hver gang. Det kan også oppstå «hull» i minnet der ferdigkjørte programmer har vært. Når flere av disse har blitt tatt ut av minnet kan man slå sammen disse «hullene» og få plass til et program til i minnet.

Dersom man skal kjøre et program som er større enn minnet må man dele det opp i segmenter. Disse segmentene legges på «swapping»-delen av disken og hentes inn ved behov. Hvert segment får da en egen base når de hentes inn i minnet. Vi får da flere slike baser som må holdes styr på. Vi sier da at vi opererer med en segmenttabell, eller en sidetabell om det er et sidedelt system vi operer med.

### Allokering av minne

Nå skal vi se på forskjellige teknikker for å hente inn programmer i minnet, såkalt allokering av minne.

**Fragmentering av minnet**

Når man flytter programmer inn og ut av minnet kan det oppstå små hull, altså biter i minnet som er for små til at hele programmer kan ligge der. Dette kalles *fragmentering* og kan løses ved *kompaktering* som vi si at man flytter alle programmer til den ene enden av minnet, og dermed samle all ledig plass i et sammenhengende område. Det å finne en løsning på fragmentering av minnet er en av operativsystemets viktige oppgaver, men kompaktering er meget tungt for systemet, så denne vil vi helst unngå.

**Faste partisjoner**

Her deles minnet inn i flere partisjoner med ulik størrelse, slik at prosesser med ulik størrelse kan stå i kø for den partisjonen som passer dem best. En ulempe med dette er at det kan bli kø for en partisjon, selv om minnet totalt sett kunne rommet alle prosessene i køen. Et enklere problem er at en partisjon har lang kø, mens en annen står tom. Dette kan løses med metoden under.

**Faste partisjoner med felles kø**

Her står prosessene i en felles kø og legges der det passer. Også her vil det oppstå fragmentering siden enhver prosess ikke vil være akkurat like stor som partisjonen den ligger i.

**Variable partisjoner**

Denne teknikken ser på minnet som et stort område og forsøker å fylle opp minnet nedenifra. Dette fyller minnet relativt godt helt til noen prosesser fullfører og det oppstår fragmentering. Denne teknikken krever dermed også kompaktering noe som ikke er optimalt.

### Teknikker for minneadministrasjon

**Swapping**

Swapping dreier seg om å hele tiden flytte prosesser mellom minnet og harddisken. På denne måten kan de mest aktive prosessene få plass i minnet, mens de prosessene som trenger for eksempel inndata, kan vente på dette mens de ligger på disken. Prosesser som ikke er klare for å kjøre på cpu trenger jo ikke ligge i minnet. På denne måten får man plass til mange prosesser i minnet, samtidig at det er de mest aktive som får plassen.

**Virtuelt minne teknikker**

Denne teknikken deler opp prosesser i deler og lar kun de mest aktive delene kjøre. Ved behov kan du mindre aktive delene hentes inn fra harddisk og legges i minnet. Denne teknikken fungerer godt med programmer som er strukturert i løkker og tabeller, dvs. Programmer der kun deler av koden er aktiv om gangen.

### Virtuelt minne

Moderne CPUer har innebygd støtte for virtuelt minne. De har nemlig en MMU-enhet som står for omregningen fra virtuelle til fysisk adresser. Derfor forholder alltid CPUen seg til virtuelle adresser.

**Sidedelt system**

I et sidedelt system deles adresserommet til en prosess inn i like store sider. På samme måte deles minnet inn i like store blokker, der en blokk kan lagre en side. På denne måten reduserer man fragmentering. Og man trenger ikke lete etter en passende plass når man skal legge noe i minnet siden alle plaser er like store. Når ny deler av prosessen kreves kan man bare importere den gjeldene siden inn i minnet. Den eneste fragmenteringen som kan skje her er *intern fragmentering*, som oppstår fordi den siste siden som allokeres i gjennomsnitt vil være halvfull. En slik side har forskjellige størrelser i forskjellige systemer, men det er vanlig med sidestørrelse på 4Kbyte.

Når CPUen nå legger en adresse på adressebussen for å hente inn neste instruksjon går denne adressen gjennom MMU-enheten. Her finnes en sidetabell som inneholder en linje for hver side som finnes i prosessens adresserom. Denne tabellen viser hvor den aktuelle siden ligger i minnet. Denne tabellen viser også om siden befinner seg i minnet eller ikke. Dette holder den styr på ved hjelp av et «flagg» som settes til 1 dersom siden er i minnet og 0 dersom siden er på harddisken.

**Segmentdelt system**

Problemet med sidedelt system er at det er vanskelig å for eksempel dele programdeler og dataområder med andre prosesser, fordi side-grensene innebærer at data som skal deles er på samme side som data som ikke skal deles. Denne modellen er optimal fordi man kun trenger å ha aktive deler av en prosess i minnet samtidig. Derfor ønsker vi ikke å kvitte oss med denne teknikken, men forbedre den. Vi deler derfor en prosess inn i segmenter som er mer tilpasset prosessens adresserom. Enkelt forklart kan vi da ha et segment for dataområdet, ett segment for stakken, ett segment for koden. På denne måten blir det enklere å sette tilgangsrettigheter til for eksempel data. Andre fordeler er at programdeler kan endres og kompileres separat. Likevel får vi fortsatt den ulempen at minnet fragmenteres og vi får problemer med å finne ledig plass til neste segment som skal inn i minnet.

I stedet for en sidetabell bruker vi nå en segmenttabell som angir for eksempel baseadressen som sier hvor et segment befinner seg. I tillegg må nå lengden på segmentet tas med siden segmentene har ulik lengde. De virtuelle adressene består nå av to deler, et segmentnummer som angir hvilket segment adressen ligger i, og en del som angir hvor langt ut i segmentet adressen er. Flagget som angir om segmentet er i minnet eller ikke er fortsatt med, sammen med et flagg som viser hvilke rettigheter prosessen har til segmentene.

**Kombinert segmentdelt sidedelt system**

Vi ønsker å trekke ut fordelene med hver av de to typene minneadministrasjon. Dette er da fordelen til sidedelt at alle enhetene er like store, og fordelen med segmentdelt der hvor enhetene bedre representerer programmets indre struktur. Segmentene kan da deles lettere med andre prosesser og tilgangsrettigheter blir enklere å sette.

Vi lager da et system med segmenter, der hvert segment er delt inn i sider. På denne måten kan man hente inn deler av segmenter i minnet, men likevel sette tilgangsrettigheter på segmentene, og dele segmentene. For å få til dette må hvert segment ha sin egen sidetabell. Dette betyr at hver prosess har en segmenttabell der hvert segment igjen har en sidetabell. En virtuell adresse består nå av 3 deler. Et segmentnummer, et sidenummer, og *offset* som angir hvor langt ut i siden den aktuelle adressen er. Siden hver referanse må innom sidetabellen, må vi også sjekke om den aktuelle siden er i minnet. Dersom den er det kan man finne frem til rett adresse.

Dersom siden ikke ligger i minnet, sier vi at vi har en *sidefeil*. Da må den aktuelle siden hentes inn i minnet før prosessen kan fortsette. Da stoppes prosessen og kjøres ikke på cpu igjen før den rikigtige siden er hentet inn til minnet, og sidetabellen viser at siden ligger i minnet.

Det er altså viktig at vi bruker effektive metoder for å skifte ut sidene i minnet, slik at vi slipper sidefeil. Dette gjelder først og fremst når minnet er fullt og operativsystemet må bestemme hvilken side den skal ta ut av minnet. Nedenfor følger noen metoder for å gjøre dette.

**Optimal**

Denne metoden er den optimale, men er umulig fordi den krever at vi kan se inn i fremtiden. Denne metoden går ut på at operativsystemet vet hvilke sider som skal brukes i fremtiden og dermed kan bytte ut den siden det er lengst til skal brukes.

**FIFO**

First in, First Out er en enkel metode. Her er det siden som kom først inn i minnet som må først ut av minnet.

**LRU**

Least Recently Used kaster ut den siden som har vært minst brukt i det siste. Dette begrunner den med at den vil bli minst brukt i fremtiden også. Det er også denne metoden som er mest i bruk. Den krever riktignok at operativsystemet holder styr på hvilke siden som har vært aktive i løpet av en periode. Dette gjøres ved at en bit settes hver gang en side brukes. Når perioden er over nullstilles alle disse bitsene. Det vil si at sider med denne biten lik 0 vil byttes ut ved sidefeil. Sidetabellen inneholder også en bit som sier om siden er skrevet til. Dette er viktig fordi da må siden skrives tilbake til disken før den kan byttes ut. På denne måten kan operativsystemet også velge å bytte ut de som ikke har denne biten satt (denne kalles «dirty» bit), siden det tar lengre tid å bytte ut slike sider.

Assosiative register er en metode for å minske referanser til minnet, siden både segmenttabellen og sidetabellen ligger i minnet. Løsningen er registere som inneholder de sidene som nylig er brukt i håp om at disse skal brukes igjen. Da kan CPU hente adressene deres fra registeret i stedet for å måtte hente den fra sidetabellen i minnet.