Oppgave 1

De fleste moderne operativsystemer utruster prosesser med privat og partisjonert minne. Dette sikrer at prosesser ikke forstyrrer hverandres data; det holder også forskjellige typer data atskilt i hver prosess. Segmentene er vanligvis navngitt som følger, med antydning til bruken: systemdatasegmentet (også kjent som prosesskontrollblokken (PCB)), kodesegmentet (eller tekstsegmentet), det initialiserte datasegmentet (eller bare datasegmentet), det uinitialiserte datasegment (også kjent som BSS-segmentet), heapen og stakken.

Vanligvis er et program først lagret på disk, hvorfra det bringes inn i minnet. Vil du argumentere for at kodesegmentet inneholder (deler av) programmet eller prosessen - eller begge to?

Kodesegmentet eksisterer ikke før ditt kompilerte program har blitt kjørt inn i terminalen. Instruksjonene som finnes i programmet ditt, er de instruksene som prosessene skal utføre. Pål hevder at man må tolke hvilken vinkling man diskuterer i forbindelse med “deler av”. For svaret er bådet et ja og nei med tanke på det jeg skrev på starten. Pål ville ha tolket det som at kodesegmentet er en del av prosessen og ikke programmet i forbindelse med memory layout. Årsaken skyldes av at kodesegmentet inneholder instrukser som er en “del av” prosessen. Derfor er kodesegmentet en del av prosessen, ikke programmet.

Heapen og stakken mot hverandre og motsatte ender av adresserommet (ettersom data lagres til dem). Hva er hensikten med denne ordningen? Hva skjer hvis rommet mellom dem blir oppbrukt?

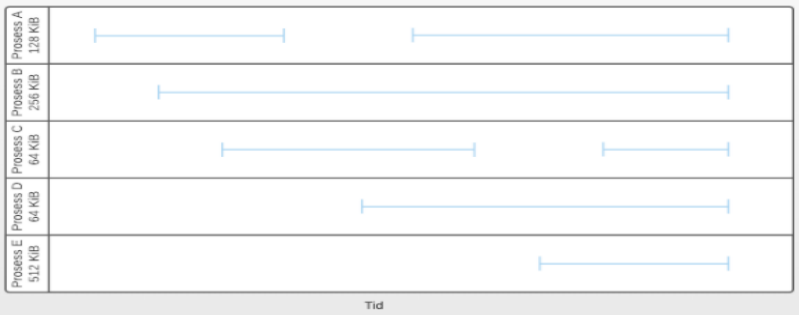
Årsaken til denne ordningen er lagt på grunn av mulighetene bak et program, altså det kan hende at et program har rekursive kall i funksjonen eller mange mallocs f.eks. Det blir lagt en del minnet for det “ubrukte minneområdet” mellom heap og stack siden som nevnt over, så vet man ikke hvordan ditt program er satt opp. Dermed blir det lagt en mel minnet som blir da utnyttet ved å la stack og heap vokser mot hverandre. Ved at de vokser mot hverandre, sørger for å effektivt utnytte minneområdet. Hvis rommet mellom stack og heap blir oppbrukt, så vil prosessen krasje siden minnet er blitt brukt.

Heapen og stakken inneholder begge prosessens kjøretidsdata. Å velge hvilken som bør brukes når er ikke alltid åpenbart for programmereren. Hvordan drar en prosess nytte av to forskjellige minnestrukturer som disse? Ville bare én være tilstrekkelig?

Da må man huske hva de ulike delene er ansvarlig for å benyttes til. Heapen er en dynamisk minneallokering hvor mallocs f.eks blir lagt hit. Imens stacks har blokker som representerer ethvert funksjonskall og oppbevarer parameterne, variabler til de ulike funksjonene og allokerer blokker for hver funksjon. Så for å dra nytte av de forskjellige minnestrukturene så må man huske at heapen brukes til dynamisk minneallokering og stacken benyttes for å oppbevare parametere til funksjonskall. Man kan tilstrekkelig ha en.

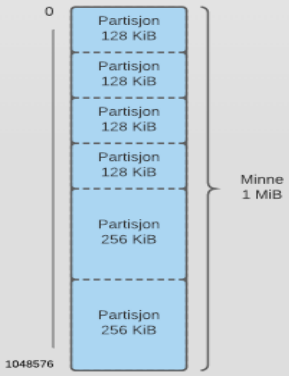
Oppgave 2

Norsk



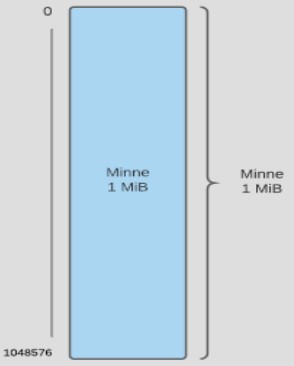
Figur 1: Kjøretidsplan

Fem programmer kjører på et meget lite system, som en værstasjon til hjemmebruk, med beskjedne 1 MiB minne. Når Program A eksekveres, kjører det i en prosess som tar opp 128 KiB. For enkelhets skyld kaller vi det Prosess A hver gang det kjører. Når Prorgam B eksekveres, tar dets prosess opp 256 KiB. Prosesser for C og D tar opp 64 KiB. Prosesser for E tar opp 512 KiB. Figur 1 viser kjøretidsplanen.



Figur 2: Statisk partisjonering

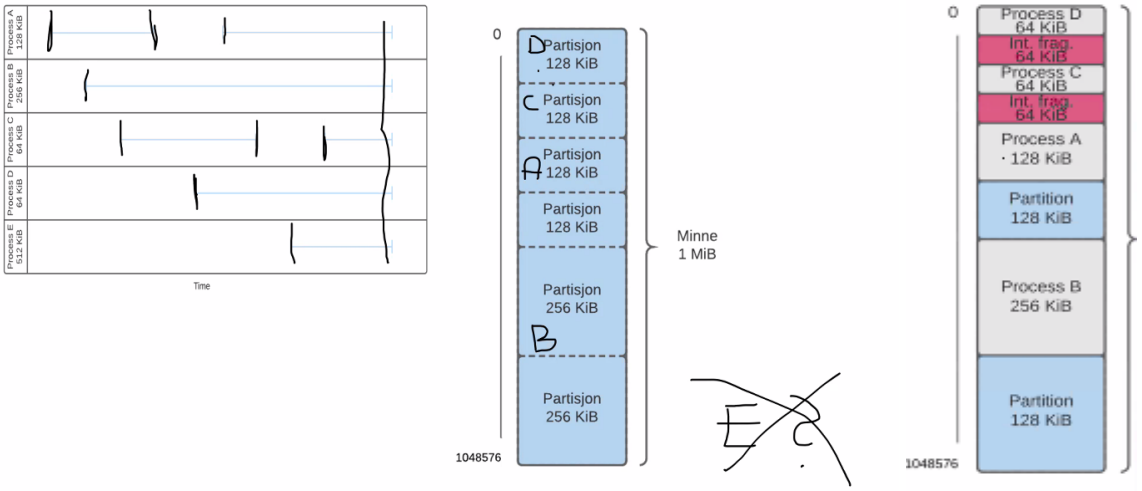
Det finnes flere metoder for å tildele prosesser minne. Dette er en viktig del av minnehåndteringen. Tre slike metoder behandles her: statisk partisjonering (med partisjoner av ulik størrelse), dynamisk partisjonering og *buddy*-tildeling. Segmentering anvendes ikke; én prosess må få plass innenfor én partisjon.



Figur 3: Dynamic partisjonering

Ta i betraktning den statiske partisjoneringsplanen vist i Figur 2. Hvordan er prosessene plassert i minnet like før slutten? Hva skjer med Proses E? Hva slags og hvor mye fragmentering forekommer?

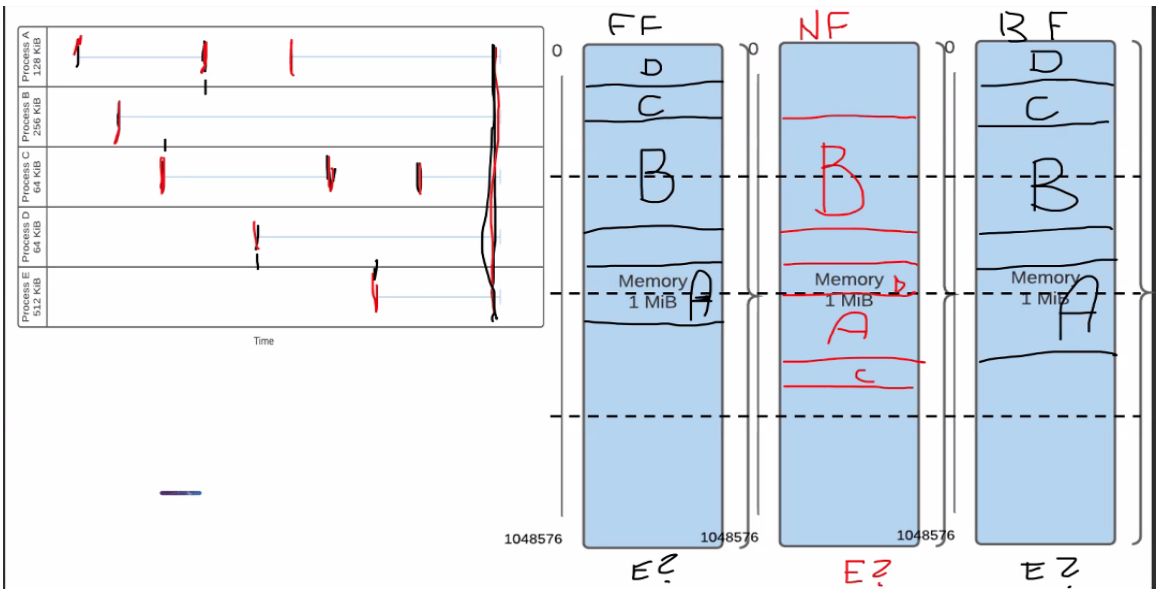
Like før slutten, er prosessene satt opp slik:



Vi ser at prosess “E”, ikke får kjørt siden vi ikke har en partisjon på plass 512. Det oppstår intern fragmentering med tanke på både “C” og “D” bruker 64 kb og vi bortkaster partisjonene som både “D” og “C” bruker. Vi får ingen ekstern fragmentering med tanke på partisjonene med “128, 256” kib, siden disse er ubrukt. Vi bruker ikke disse partisjonene.

Figur 3 viser minnelayouten uten forhåndsbestemte partisjoner. Dette støtter *dynamisk* partisjonering. Hvordan er resultatet annerledes når dynamisk partisjonering brukes med hver av tildelingsalgoritmene *First Fit*, *Next Fit* og *Best Fit*? Hvilken mekanisme kan hjelpe Prosess E? Når man velger en tildelingsalgoritme, hvilke andre kriterier – foruten fragmentering – er verdt å ta i betraktning?

Først må man forklare de ulike tildelingsalgoritmene slik at sensor skjønner at du forstår disse algoritmene og hvordan det er satt opp ved å vise hver av tilfellene. 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” leter fra da man sist allokerte et segment og som first fit, finner da den beste mulige plassen/partisjon. “Best-fit” leter gjennom hele lista av partisjoner som er ledig og finner den som er best. “Best” vil også velge den partisjonen som leder til minst fragmentering.

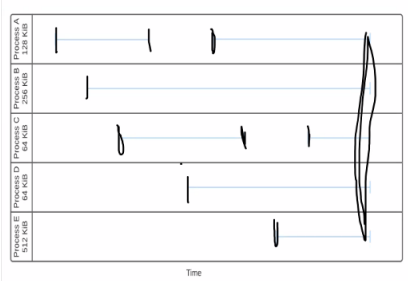


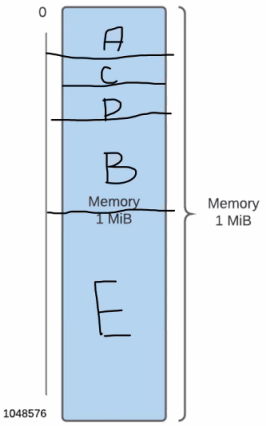
Dette bildet over, er med på å beskrive de ulike tildelingsalgoritmene hvor vi da setter inn prosessene fra oppgave 2. Kan være vanskelig å tolke hva som blir sett her, så anbefaler og ser videoen igjen. Men vi ser at prosess “E” ikke har plass til å bli kjørt hvor ingen av mekanismene vi har sett, vil hjelpe denne prosessen.

Andre kriterier som man burde ta i betraktning, er kompleksiteten bak å lage en slik algoritme ved de tre tildelingsalgoritmene, samt kostnaden bak en tildelingsalgoritme som “best-fit”. Vi husker ved notatene om dette at, “best-fit” kan skape fragmenter som blir såpass smale at disse ikke blir benyttet i det hele tatt. Så “first-fit” og “next-fit” er bedre alternativer enn “best-fit”.

***Buddy*-tildeling henter inspirasjon fra både statisk og dynamisk partisjonering – som en hybrid mellom dem. Hvilket resultat gir *buddy*-tildeling?**

Først til å forklare buddy-tildelingen: 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.





Med buddy-systemet så vil vi få plass til alle prosessene. Alle prosessene vil bli utnyttet i minnet. Viktig å merke at buddy-systemet ikke fungerer alltid, men i dette tilfellet med tanke på størrelsene vi hadde fått og kjøretidsplanene ble gitt slik, så funket dette.

Oppgave 3

For minnehåndtering gir virtuelt minne flere fordeler. Applikasjoner har tilgang til et kontinuerlig adresserom, hvorav forskjellige deler kan være tilordnet minne eller disk. På denne måten kan mengden virtuelt minne langt overstige mengden fysisk minne tilgjengelig. Kostnaden er selvfølgelig at diskaksesser er tidsmessig dyrere; diskaksesser er flere størrelsesordner langsommere enn minneaksesser.

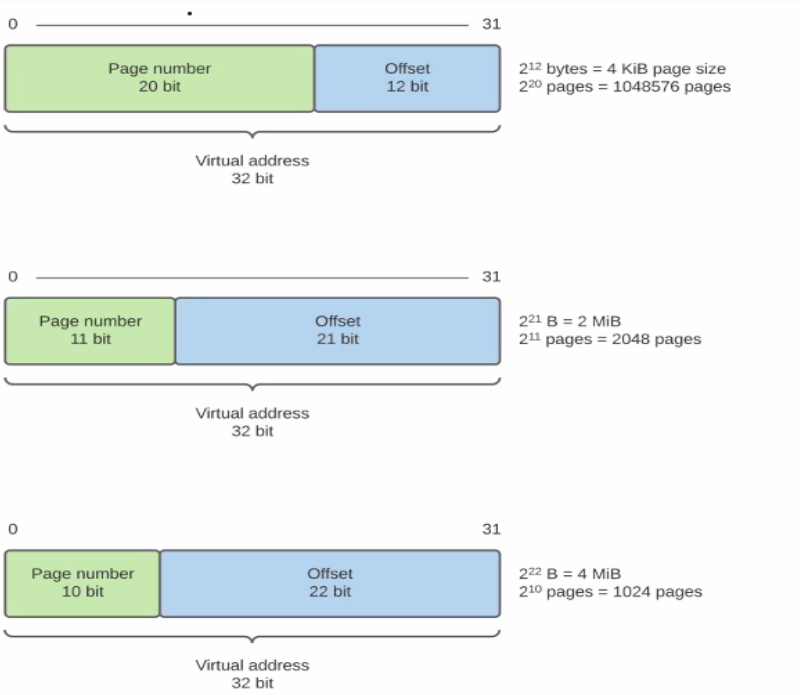
Side	Ramme	Til stede
111	000	1
110	101	1
101	100	0
100	110	1
011	010	1
010	011	1
001	111	0
000	001	1

Figur 4: Sidetabell

På IA-32 er det virtuelle adresserommet 2^{32} B = 4 GiB. Sidestørrelsen er vanligvis 4 KiB, men kan være 2 MiB eller 4 MiB (under henholdsvis *Physical Address Extension* (PAE) og *Page Size Extension* (PSE). Hvordan avgjør sidestørrelsen antallet sider tilgjengelig? Hvordan er virtuelle adresser sammensatt for å støtte hver av disse sidestørrelsene? Hvordan relateres sidestørrelse til rammestørrelse?

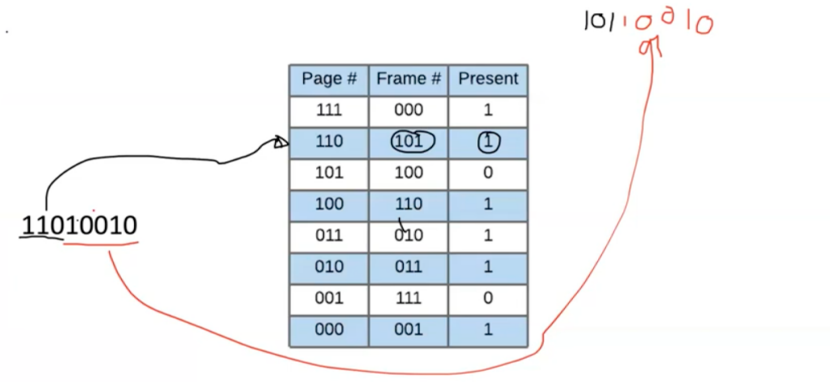
Hvordan man regner ut offset basert på kib fra sidestørrelsen, i dette tilfellet 4 kb.
2 opphøyd i 10 er 1024 = 1 kb
så har du 2 bits til, altså 1024*2*2 = 4096=4k

2^12 = 4kb



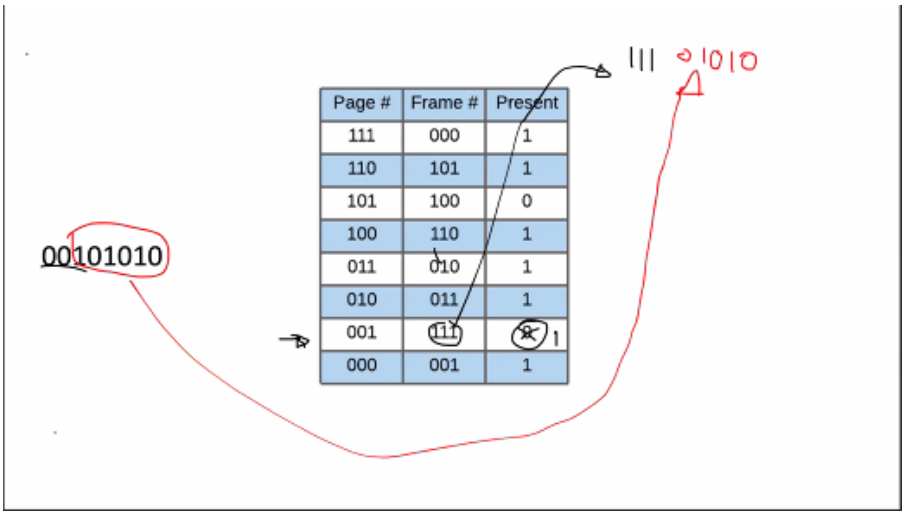
Til å finne ut hvilken KIB-størrelse som er aktuelt får hver page, så er det avhengig av systemet som er blitt satt. Har vi filer i systemet som krever mange blokker av data, så kan 4kb størrelse være aktuelt siden vi har plass til mange pages. Men har vi et system som ikke behøver så mange pages, så er det unødvendig å allokere så mye plass til pages for da får vi mye unødvendig intern fragmentering.

Minneoppslag, altså oversettelsen mellom virtuelle og fysiske adresser, utføres av minnehåndtereren (engelsk: *Memory Management Unit* (MMU)). Ta i betrakning sidetabellen (simplifisert) vist i Figur 4. Hvordan oversettes den åttebits virtuelle adressen 11010010 til sin fysiske motpart? Hvordan håndterer systemet en forespørsel etter adressen 00101010? Gi en sammenfatning hendelsesforløpet.



Hvordan oversettes den åttebits virtuelle adressen 11010010 til sin fysiske motpart?

Når man skal løse slike oppgaver, så må man først lese de 3 første bitene, som referer til hvilken adresse som pagen befinner seg i det fysiske minnet. Deretter leser vi minnerammen som er oppgitt og observerer present-bittet for å sjekke at vi ikke noen prosesser som må aksesseres fra disk. Da blir den virtuelle adressen “11010010” gjort om til “10110010” hvor “101” er framen og offset er “10010”.



Ved adressen “00101010”, så har vi at present-bittet er satt til 0. Dermed må vi aksessere disken og håndtere denne “page-faulten”. Dette gjøres ved at vi switcher om en prosess i kjernen med en i systemet. Først må vi forsikre om at prosessen i systemet har blitt oppdatert for å ivareta dets data. Deretter vil OS´et finnet prosessen som blir offeret, altså den som befinner seg i systemet, og skedulerer en diskoperasjon som har ansvar om å sette prosessen inn i minnet. Når operasjonen er ferdig, får man et interrupt hvor page-tabellen blir oppdatert hvor man markerer pagen til å settes til “normal state”. Present-bittet vil endres til 1 gjennom “normal staten” og da kan vi utføre “Memory lookup” igjen.

En sidefeil inntreffer når den forespurte siden ikke finnes i minnet. En sideerstatning inntreffer når det ikke finnes noen fri side som tilfredsstiller minnetildelingen. Det finnes flere sideerstatningsalgoritmer, som FIFO, *Second Chance* og *Least Recently Used*. Viktige ytelsesindikatorer for slike algoritmer er hyppigheten av sidefeil og tidskompleksiteten. Hvordan lar disse algoritmene seg sammenlikne etter slike målestokker?

Vi forklarte i spørsmålet over at, når vi har en sidefeil, så må vi aksessere prosessen fra systemet som skal bli “offeret” til prosessen i kjernen. Vi har da ulike erstatningsalgoritmer som FIFO, Second-Chance og Least-Recently-Used. Ved FIFO, så kan det forekomme overheads ettersom vi aksesserer disken og kan bli problematisk hvis hyppigheten av sidefeil forekommer ganske ofte. Problemet er om vi har et scenario hvor prosess “A” ble kjørt ferdig og lagt inn i disken også skal denne prosessen gjenta seg igjen senere i omløpet.

Da vil vi få overhead som nevnt over og dette kan bli problematisk hvor vi har prosesser som gjentar seg etter at de er blitt kjørt ferdig siden disse prosessene blir lagt inn i disken og vi må da aksessere disken hver gang dette skjer.

I Second-Chance så er dette en forbedring ved FIFO hvor vi ser på referansebittet til hver prosess og de som har “1-bit” som skulle opprinnelig ha blitt kastet ut av bufferet og lagt inn i disken, vil bli lagt først i køen og få kjøre sin prosess igjen. Dermed slipper vi å måtte aksessere til disken, men det å bytte om prosesser innad i bufferet på en slik måte er fortsatt kostbart. Så igjen hvis det forekommer sidefeil ofte så må vi da utføre operasjoner hvor vi bytter om prosessene i bufferet og vil da være meget kostbart, men bedre enn FIFO.

Ved Least-Recently-Used så er dette den beste alternativet av de tre. Viktig å merke at det finnes ulike implementasjoner av algoritmen og min forklaring baseres på implementasjonen som er gått igjennom i forelesningen gjennom en lenkeliste. Prosessene er satt opp slik at, de som forekommer sjeldent, vil plasseres mot slutten av listen også vil de som forekommer ofte, bli satt på starten av listen. Dette fører til at vi minimerer tilfeller hvor disken blir aksessert siden prosessene som gjentar seg, nødvendigvis ikke blir kastet ut, men satt lenger fremme i køen. Så hvis sidefeil opptrer hyppig, så vil ikke dette være et stort problem ved denne algoritmen sammenlignet med de over.