Eksamen 2017 Høst

1. **Et operativsystem av typen mikro-kjerne:**

* Kjører alle programmer som tråder under en prosess.

1. **En fordel med å bruke prosesser fremfor tråder i et program er:**

* Hvis en prosess krasjer, så krasjer ikke hele programmet.

1. **Årsaken til at et program oppretter flere tråder i en prosess er:**

* Å gjøre flere ting samtidig, samt å dele ressurser innen prosessen.

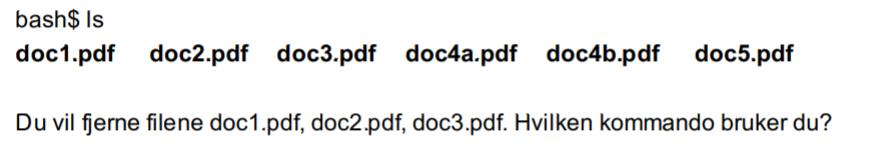
1. **Prosesstabellen inneholder blant annet følgende informasjon:**

* CPU-registrer, instruksjonspeker, prosess ID, foreldre-prosess ID

1. **Hensikten med å skille brukermodus fra kjernemodus er:**

* å ikke tillate brukerprogrammer direkte tilgang til maskinvaren.

1. **Du har en mappe med innholdet:**



* rm doc{1..3}.pdf

1. **Beskriv hva systemkallene fork, exec, og wait gjør.**

**Hva er forholdet mellom disse tre?**

**I hvilken rekkefølge bør de brukes i et skall-program?**

**Fork** brukes for å lage en ny prosess ved å kopiere en annen prosess. Denne prosessen får eget plass i minnet, men vil ha akkurat likt innhold som «forelde-prosessen» like etter koperingen.

**Exec** brukes for å erstatte programmet i en prosess med et annet program, og starter kjøringen av dette programmet.

**Wait** brukes for å vente til en prosess har forandret status. Altså hvis den f.eks. går fra å kjøre til å være suspendert eller ikke-klar.

Disse tre systemkallene brukes for å oprette en ny prosess, starte kjøringen av prosessen, og vente på at dette blir gjort. I et skall-program bør de brukes i denne rekkefølgen: fork, exec, wait. I denne rekkefølgen vil man først opprette en ny prosess med fork, deretter vil man starte kjøringen av denne prosessen med exec. I tillegg brukes wait av «forelde-prosessen» slik at den venter til at prosessen under blir ferdig.

1. Anta at du skal skrive et program som leser innholdet til en fil av ukjent størrelse inn i minnet. Du kan anta at den maksimale filstørrelsen ikke er større enn det fysiske minnet på datamaskinen.

Se på listen over Linux-systemkall i innledningsteksten som står øverst og til venstre.

Hvilke av disse systemkallene bruker du, og hvorfor? Vær nøye med å angi rekkefølgen av systemkallene. Du trenger ikke å skrive C-kode, det er nok å beskrive hvert enkelt systemkall med en angivelse av hva systemkallet gjør og hvordan det brukes i programmet.

9.

**Beskriv hva som er problemet med minnefragmentering når flere prosesser opptar plass i minnet. Legg vekt på hva som gjør at minnefragmentering oppstår. Det eksisterer flere metoder for å løse dette problemet. Beskriv én løsning, og diskuter fordelene og ulempene til denne løsningen.**

Hvis minnet er helt tomt, altså ved oppstart, vil prosesser legges sekvensielt etter hverandre i minnet slik at hele minnet kan bli brukt. Dette er ønskelig å kunne gjøre hele tiden når prosesser legges i minnet, men når flere programmer har kjørt en stund vil det oppstå «hull» i minnet. Når det oppstår slike «hull» i minnet sier man at minnet er fragmentert. Da vil man oppleve at prosesser får problemer med å finne plass i minnet. Prosesser vil da ligge overalt i minnet med mindre «hull» imellom som ikke er store nok til andre prosesser.

OS kan løse dette ved å iblant defragmentere minnet, men dette er ressurstungt for OS derfor ønsker man å unngå det. En metode for å unngå minnefragmentering er sidedelt minneadministrasjon. Minnet deles da opp i like store sider, i tillegg til det virtuelle minnet. Dette gjør at man kan enkelt legge inn «blokker» minnet når de trengs. Informasjon om hvor prosesser ligger i minnet er lagret i en sidetabell i MMU, hvor adressekonvertering fra virtuell til fysisk minneadresse foregår.

Ulempen med sidedelt minneadministrasjon er at minnet vil aldri bli helt optimalisert brukt. Hvis en prosess ikke bruker en hel blokk vil man bruke mer minneplass enn nødvendig.

1. **Hva er korrekt heksadesimal verdi av følgende binære verdi: 101101**

* 2D

1. **Hvilken eller hvilke av de binære tallene nedenfor tilsvarer verdien til det heksadesimale tallet: AF.**

* 000010101111 og 10101111

1. **Ta utgangspunkt i følgende binære verdi: 100000 Nedenfor finner du noen ulike tall som enten er på heksadesimal eller på desimal form. Hvilket eller hvilke av disse tallene kan tilsvare den binære verdien:**

* 32 på binær
* 20 på heksadesimal

1. **Hva er et lagerhierarki? Vis et slikt hierarki, og beskriv hvert enkelt nivå. Diskuter de viktigste egenskaper og karakteristika for hvert nivå.**

Et lagerhierarki er et hierarki med forskjellige lagringsmekanismer med ulike egenskaper. I et lagerhierarki er det noen prinsipper som generelt gjelder:

* Jo hurtigere lagertype, jo mer koster per bit.
* Jo høyere lagringskapasitet, jo billigere per bit.
* Jo høyere lagringskapasitet, jo langsommere er den (aksesstiden).

Et eksempel på et lagerhierarki som finnes i de fleste datamaskiner i dag:

På det nederste laget finner man harddisker. Disse lagringsenhetene er de tregeste i dette hierarkiet, men også de billigste per bit. Harddisker måles ofte opp i flere hundre GB, altså flere hundre milliarder bit.

Ovenfor harddisker finner man ofte SSD (Solid State Drive). Disse er meget kjappe i forhold til magnetiske disker som harddisker, men derfor også en del dyrere per bit. SSD-er lagrer informasjon slik at det er ingen mekaniske deler involvert. Man har sett en utvikling innenfor datautstyr hvor komponenter med mekaniske deler blir erstattet av raskere komponenter med elektriske deler. Et eksempel på dette er blant annet når reeler ble byttet ut med transistorer.

Etter dette finner man primærminnet, eller RAM (Random Access Memory). Denne lagrinstypen er en såkalt DRAM. Den er meget kjapp i forhold til lagrene beskrevet under i hierarkiet. Etter hvert som man beveger seg oppover i hierarkiet blir det dyrere per bit, og derfor blir størrelsen på lagringen mindre og mindre. Primærminnet måles ofte i noen GB. Servere og kraftigere maskiner kan ha oppimot flere hundre, kanskje til og med tusen, GB med RAM. Så mye RAM er ikke vanlig å ha i en normal maskin grunnet pris og nødvendighet.

Videre så finner man cache. Cache ligger ofte i flere nivåer inni CPU-en, disse kalles ofte L1, L2, L3 osv.. Cache er en lagringsenhet av typen SRAM. Disse skiller seg ifra DRAM ved at de lagrer bit/informasjon på en annen raskere måte. Cache brukes i forbindelse med prinsippet om lokalitet for å lagre nødvendig informasjon som man tror blir brukt senere. Ved å lagre dette i en raskere lagring vil man kunne utføre instruksjoner raskere. Størrelsene på disse er relativt små i forhold til de andre lagringene beskrevet. Ofte kan den være f.eks. 8- eller 64 byte.

Den siste lagringstypen i hierarkiet er registre som ligger i CPU-en. Disse registrene brukes under kjøring av instruksjoner. Disse må være utrolige raske, siden de må holde følge med hastigheten på CPU-en. Nå i dag kan prosessorer utføre flere milliarder instruksjoner hvert sekund, da er man nede i nanosekunder for hver instruksjon. Registre lagrer oftest bare et ord, med en størrelse på f.eks. 64 bit eller gjerne mindre.

Dette er et meget vanlig hierarki i datamaskiner nå i dag, med ulike konfigurasjoner av lagrinstypene beskrevet. Man kan også se at prinsippene nevnt ovenfor gjelder i dette hierarkiet.

1. **Hva sier prinsippet om lokalitet? Begrunn hvorfor prinsippet gjelder. Gjelder prinsippet om lokalitet mellom andre deler av lagerhierarkiet enn mellom register og minne? (Begrunn svaret)**

Prinsippet om lokalitet sier at data og instruksjoner nært den som trengs vil mest sannsynlig bli brukt i nær fremtid. Altså hvis CPU må ut i primærminnet for å hente data, da kan den ta med seg en blokk slik at den slipper å gå ut i primærminnet etter hver instruksjon. Denne blokken vil da bli lagret i cachen slik utføringen av instruksjonene går fortere, man trenger ikke å vente på aksesstiden som henting i primærminnet tar.

Dette prinsippet gjelder også for andre deler av hierarkiet. Et prinsipp innenfor lagerhierarkiet er at jo lengre ned man kommer i hierarkiet, jo sjeldnere aksesseres det av CPU. Altså hvis man allerede må ut i en tregere lagrinstype, da kan man ta meg seg andre data som mest sannsynlig kommer til å bli brukt i nær fremtid. F.eks. mellom primærminnet og harddisk. Harddisken har en buffer som er raskere enn selve harddisken med data som ofte brukes, dette kalles ofte SWAP området. Grunnen til dette er programkode ligger på SWAP området eller bufferet, og legges inn og ut i primærminnet ettersom det trengs.

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

Cachen lagrer data i såkalte «blokker». Hvordan man organiserer aksesseringen av disse blokkene har mye å si for aksesstiden til cachen. En cache kan ha tre forskjellig mapping metoder:

**Full assosiativ:** En blokk kan legges i hvilken som helst linje i cachen. Størrelsen på disse linjene angis i linjestørrelsen som f.eks. kan være 64 byte. Da må CPU lete igjennom alle linjene til den finner blokken, eller absolutt alle linjene hvis den blokken ikke er i cachen. Dette er lite effektivt selv med den hurtigste elektronikken nå i dag. Dette vil også øke aksesstiden til cachen som ikke er ønskelig, derfor brukes ikke denne metoden.

**Ikke assosiativ (direkte mapping)**: En blokk «tilhører» en linje. F.eks. linje 1 kan kun inneholde blokk 1, 3 og 5, og linje 2 kan kun inneholde blokk 2, 4 og 6. Da trenger ikke CPU å sjekke alle linjene, men kun den spesifikke linjen som inneholder den blokken. Elektronikk vil vite hvilke blokker som tilhører hvilken linje. Ulempen med denne metoden er når to blokker tvinger hverandre ut av cachen hvis de tilhører samme linje. En linje kan kun inneholde en blokk, derfor vil den forsvinne hvis en annen blokk som tilhører samme linje blir lagt i cachen. Dette kalles «trashing». Denne metoden er mer effektiv enn full-assosiativ, men man kan oppleve lengre aksesstid hvis «trashing» oppstår ofte, derfor brukes heller ikke denne metoden i moderne cache.

**Sett-assosiativ**: En blokk kan «tilhøre» en sett av linjer. Altså en blokk vil tilhøre et sett på lik måte som at en blokk tilhører en linje i ikke-assosiativ, men i sett-assosiativ kan et sett inneholde flere linjer. Antall linjer per sett kan f.eks. være «8-veis» hvor det er 8 linjer per sett. Ved å gjøre dette har man til en viss grad eliminert problemet med «trashing». Derimot så kan «trashing» oppstå hvis man har få linjer per sett.

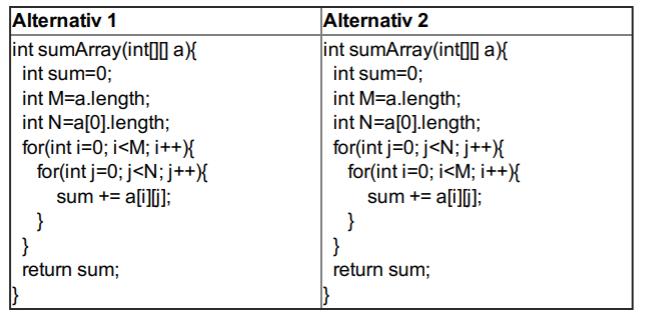
**L1**: Angir hvilket nivå cache det er snakk om, en CPU inneholder gjerne oppimot 3 ulike cacher. Disse har forskjellige størrelser.

**8-veis**: 8 linjer per sett i en sett-assosiativ cache

**Sett-assosiativt**: Blokker «tilhører» sett med flere linjer.

**Linjestørrelse (64 byte)**: Størrelsen på linjene i cachen. 64 byte vil si at hver linje kan inneholde data på en størrelse med 64 byte.

1. **Nedenfor ser du to ulike metoder i java. Begge metoder returnerer summen av alle elementene i den todimensjonale tabellen a. Begge metoder vil gi samme sluttresultat. Forskjellen på metodene er rekkefølgen elementene summeres i.**



**Spørsmål:**

* **Hva er hovedforskjellen på virkemåten til de to metodene?**

Hovedforskjellen mellom disse to metodene er måten de legger sammen alle elementene.

Alternativ 1 vil gå gjennom hvert element «sub-array» sekvensielt, før den går til neste «sub-array» også videre. F.eks. [[a1,b1], [a2, b2]] vil bli lagt sammen slik: a1 + b1 + a2 + b2.

Alternativ 2 vil derimot gå gjennom det første elementet i hvert «sub-array», før den går videre til det andre elementet i hver «sub-array» også videre. F.eks. [[a1,b1], [a2, b2]] vil bli lagt sammen slik: a1 + a2 + b1 + b2.

* **Hvilken av de to metodene vil utnytte cachen best, og dermed ha den korteste utføringstiden? Begrunn svaret nøye, og beskriv eventuelle forutsetninger for svaret**.

Hvilken metode som vil utnytte cachen best avhenger av hvilke data som ligger i cachen. Hvis man antar at prinsippet om lokalitet gjelder, vil man tro at dataene legges til sekvensielt, altså hvert «sub-array» legges til etter hvert i kjøringen.

Da vil alternativ 1 være den beste metoden. Man kan legge til alle elementene sekvensielt i hvert «sub-array», enn å måte hente hvert «sub-array» i cachen, for å så kun bruke et element i hvert av de.

1. **I kurset har vi lært at overføring fra primærminne til cache foregår ved en såkalt burst.**
2. **Hva er en burst?**

Når man skal overføre en mengde data over minnebussen er man begrenset av størrelsen på bussen, altså bredden (parallell buss). Hvis bussen er f.eks. 64 bit, må man overføre 8 ganger hvis man ønsker å overføre 64 byte (1 byte = 8 bit).

Det er derfor en fordel å sende over alle dataene rett etter hverandre når man har funnet de i minnet. Det er dette som kalles en burst. Først må man vente på at det finnes, men så sendes det over uten store forsinkelser, utenom at man må vente på hver klokkepuls.

1. **Skisser hva som skjer i en burst. Legg vekt på hva som skiller en burst fra å hente det samme antall enkeltlokasjoner.**

Når CPU vil overføre data fra minnet, vil den først sende en adresse til minnet. Minnet bruker da litt tid (aksesstid) på å finne denne startadressen. Etter dette vil den sende alle dataene som ligger sekvensielt etter startadressen. Man trenger ikke sende adressen til hver lokasjon til dataene hver gang man skal overføre. Man kan dele opp prosessen i en burst i tre hovedsteg:

1. **Overføring av startadresse til minnet:** En startadresse overføres til minnet slik at minnet vet hvor den skal starte overføringen. Minnet vil da også vite når den skal stoppe overføringen, siden det er blokker som overføres. Når man har overført siste data i blokken, stoppes bursten.
2. **Vente på at minnet finner data:** Aksesstiden til minnet vil være tiden man må vente på at minnet leter etter dataene på angitt startadresse. Det er viktig å merke seg at klokkefrekvensen på minnebussen (mulige overføring per sekund målt i hz) ikke har noe å si for aksesstiden, utenom at man må vente på starten av en klokkepuls før man kan begynne å overføre etter at dataene er funnet. I DDR (Double data rate) kan man overføre to ganger per klokkepuls, derfor vil det være annet en med kun en overføring per klokkepuls.
3. **Kontinuerlig overføring**: Etter at startadressen er funnet trenger ikke minnet å lete lenger etter data, og CPU trenger ikke å sende flere adresser. Data blir overført sekvensielt kontinuerlig for hver mulig klokkepuls. Under denne overføringen har derfor klokkefrekvensen mye å si for tiden det tar å overføre dataene. Det er også derfor det er ønskelig å skru opp klokkefrekvensen så høyt som mulig uten at det oppstår praktiske problemer.