***Oppgave 1: Funksjonskall vs Systemkall***

## 1. Funksjonskall vs systemkall

### Norsk

Systemkallet `read()` forsøker å lese `count` bytes inn i bufferen pekt på av `buf` fra filen tilknyttet den åpne filendeskriptoren `fd`. Ved suksess returnerer den antall leste bytes. Ved feil returnerer den negativ én.

```
ssize_t read(int fd, void *buf, size_t count);
```

Som nesten alle systemkall i Linux, aktiveres `read()` gjennom en bibliotekfunksjon med samme navn. Det finnes også andre bibliotekfunksjoner som er avhengige av `read()`. Hvis dette virker forvirrende, les man-sidene `man 2 read` (systemkall) og `man 3 read()` (biblioteksfunksjon).

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

Systemkall fungerer som inngangspunkter til kjernen, hvor selve «tungløftingen» av for eksempel lesing av bytes fra en fil lagret på disken inn i RAM utføres. Kjernefunksjonen som tilsvarer `read()` er `sys_read()`.

**Hvis man ser bort fra forkunnskaper, finnes det noen generell måte å avgjøre om en signatur tilhører et systemkall eller en biblioteksfunksjon?**

Begge signaturene ser identiske ut. Uten å kjenne til funksjonen fra før så kan man ikke vite om en signatur tilhører et systemkall eller en biblioteksfunksjon. Hadde signaturene hatt annerledes parametere så ville de ikke vært like identiske, men i dette tilfellet så er signaturene ikke det.

**Uansett hvordan det kan se ut på overflaten fungerer funksjonskall og systemkall forskjellig internt. Systemkall utsteder TRAP-instruksjon for å gå inn i kjernen, mens funksjonskall utsteder CALL-instruksjonen for å kalle en annen funksjon. På hvilke to grunnleggende måter er oppførselen til disse instruksjonene forskjellig?**

TRAP ber om å få tilgang til kjernen, mens CALL ønsker å kalle en annen funksjon som er definert i et bibliotek med andre funksjoner. TRAP brukes i en kernel-space kontekst, mens CALL benyttes. TRAP er tilknyttet kontekst-svitsjing. Dette vil bli mer forståelig i senere forelesninger når vi skal lære om fil-lesing. Forskjellen er som nevnt at TRAP-instruksjonen gjør at man kommer ned i kjernen til å utføre de priviligerte funksjonene ved kjernen.

**Kjernefunksjoner som er pakket inn i systemanrop og pakket inne i bibliotekfunksjoner… Hvorfor ikke spare bryderiet og kalle sys_read() direkte fra**

**programmet? Er det mulig? I så fall, eller hvis det var det, ville stress spart virkelig vært større enn stress skapt.**

Vi husker dette med privilegienivåer og hvilke rettigheter som er tilgjengelig for applikasjonslaget, eller kjernen f.eks. Derfor å kalle sys_read() direkte, så vil ikke det være lov ettersom applikasjonslaget(laget som vi brukere kan styre), ikke har tilgang fra kjernen til å utføre dette. Det er mulig å gjøre om privilegienivåene til å tillate gitte lag som applikasjonen til å få lov til å benytte av kjernen. Hvis man gjør dette på bakgrunn av at dette er lov, så kan det skje mye komplekst som en ikke har kontroll over. Prosessoren kan krasje, instrukser kan gå i mot hverandre osv. Derfor er dette uønskelig og ikke anbefalt å gjøre.

## ***Oppgave 2: First In, First Out vs Shortest Job First***

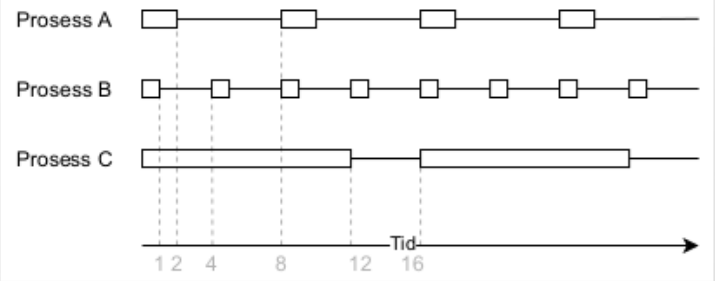
### 2. *First In, First Out vs Shortest Job First*

#### Norsk

Du er ansatt for å designe et operativsystem for hovedserveren i et sensornettverk. Tre prosesser kjører i systemet: Prosess A og B mottar kontinuerlig data fra to sensorgrupper, utsetter dataene for lett prosessering og skriver dem til disk. Prosess C leser disse dataene fra disk, beregner noen aggregater (beregninsintensiv oppgave) og skriver dem tilbake til disk. Prosessene kjører i uendelig løkke, men blokkerer for inn-/utdata. Figur 1 viser utbruddsmønstre for hver prosess. Tiden mellom utbruddene brukes på diskoperasjoner.

Å velge riktig skedulerer er en viktig beslutning. Maskinvarebegrensninger forordner at skeduleringsalgoritmen må være *ikke-preemptiv*. Du vurderer to alternativer: *First In, First Out* (FIFO) og *Shortest Job First* (SJF).

- Gitt ankomstsekvensen CAB, hvordan vil gjennomsnittlig ventetid og fullføringstid (i løpet av én iterasjon) variere mellom FIFO og SJF?
- Finnes noen vektige grunner til å velge det ene alternativet fremfor det andre?
- Hvilke ytelseskriterier bør anvendes på skedulereren for et slikt operativsystem?




Figur 1: Utbruddsmønstre for tre prosesser på hovedserveren i et sensornettverk

**Gitt ankomstsekvensen CAB, hvordan vil gjennomsnittlig ventetid og fullføringstid (i løpet av en iterasjon) variere mellom FIFO og SJF?**

Vi skal da se på den første iterasjonen, ikke hele iterasjonen i seg selv.

## Oppgave 2

a) First In, First Out: Om en benytter seg av en FIFO kø, vil ankomsten CAB gjøre at hele prosessen tar totalt 40 sekunder (2⅓ sek for å utføre C, 8 sek for å utføre A og 8 sek for å utføre B). Den gjennomsnittlige ventetiden for hver prosess blir da: 18,67 sek. Den gjennomsnittlige fullføringstiden blir: 32 sek.

| | Trenger | Ventetid | Ferdig |
|---|---|---|---|
| A | 8 sek | 24 sek | 32 sek |
| B | 8 sek | 32 sek | 40 sek |
| C | 24 sek | 0 sek | 24 sek |

Shortest Job First: Benytter en seg av SJF, vil hele prosessen ta like lang tid – 40 sek. Men den gjennomsnittlige starttiden vil være lavere – 8 sek. Den gjennomsnittlige fullføringstiden vil også være lavere, nemlig 21,3 sek.

| | Trenger | Ventetid | Ferdig |
|---|---|---|---|
| A | 8 sek | 8 sek | 16 sek |
| B | 8 sek | 0 sek | 8 sek |
| C | 24 sek | 16 sek | 40 sek |

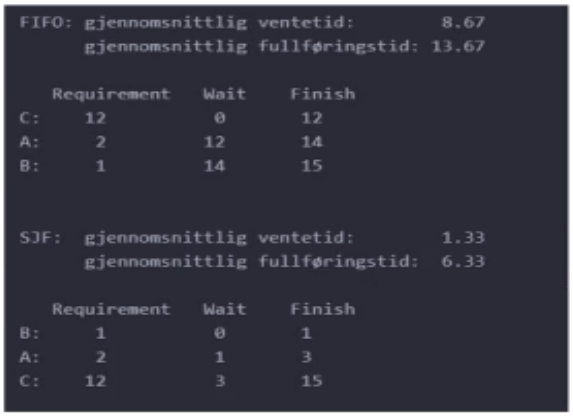
Det blir altså kortere ventetid og fullføringstid i snitt ved å benytte seg av Shortest Job First.

**FIFO**

Problemet ved oppgavebesvarelsen når det gjelder over ligger i at denne gruppen har manglet å forklare hvordan de tenkte ved regneoperasjon av **FIFO**. I tillegg til at tiden "C" bruker på å bli ferdig med sin prosess, så har gruppen tenkt slik at "C" må fullføre hele sin del, og da skal "A" få lov til å utføre sin del og tilslutt "B". Etter sin første iterasjon så er "C" ferdig og da skal "A" og "B" få mulighet til å kjøre på bakgrunn av at det er rom for både "A" og "B" til å utføre sin prosess. Dette kommenterte Pål på og derfor så skulle gruppen ha begrunnet at de så på ankomstsekvensen slikt, altså at hele "C" må bli ferdig med sin prosess først, deretter "A" og tilslutt "B". Da ville Pål ha godtatt denne besvarelsen ettersom gruppen hadde begrunnet med å si at: "Vi har antatt at, på bakgrunn av ankomstsekvensen så vil vi først fullføre iterasjonen av hele "C", så "A" og tilslutt "B" ". Da hadde gruppen fått helt riktig i oppgavebesvarelsen sin.

Gruppen under sin besvarelse er derfor Pål mer enig med ettersom de tok hensyn til ventetiden til "C". Men begge svarene er som sagt riktige, er bare mangel på begrunnelser i gruppen over imotsetning til gruppen under. Kan også sies om gruppen under også ettersom de mangler å begrunne noe tekst bak utregningen sin også.

a)




Her ser man at gjennomsnittlig ventetid og fullføringstid er høyere med FIFO som skeduleringsalgoritme enn SJF.

**SFJ**

Forskjellen mellom FIFO og SJF i denne konteksten, er følgende: Ettersom vi velger ut de prosessene som kjører først, så vil “B” kjøre først, deretter “A” og da kommer “B” igjen ved første iterasjon. “C” må da vente en stund til å utføre sin prosess, noe som blir begrunnet mer i neste spørsmål. Bildet over har da gjort riktige beregninger når det gjelder første iterasjonen.

**Finnes noen vektige grunner til å velge det ene alternativet fremfor det andre?**

Her er det viktig å huske prinsippene ved FIFO(First in, First out) og SJF(Shortest Job First) når det gjelder begrunnelsen. I tillegg til at scheduler-algoritmen må være non-preemptive. Dvs, et scheduler-system hvor scheduleren håndplukker ut prosesser ved å ikke fokusere på prioriteten til hver eneste prosess.

**Hvorfor velge FIFO?**

Det finnes flere grunner til å velge FIFO fremfor SJF i denne konteksten. Når det kommer prosessorer så må SJF sortere etter de som tar kortest tid å kjøre, noe som er kostbart og mer komplekst enn FIFO. FIFO er ikke like komplisert ettersom scheduleren setter inn prosesser og kjører de som kom først. Dermed er kompleksiteten og kostbarhet ved SJF en grunn til å velge FIFO istedenfor.

En annen grunn er å se på kjøretiden for gitte prosesser i “CAB”-sekvensen, spesifikt “C”. Ved en SFJ, så må “C” vente mot slutten ettersom både “A” og “B” har kortere tid til å utføre sine prosesser i motsetning til “C”. Derfor må “C” vente lenge inntil den får lov av scheduleren til å utføre sitt program. Det at en prosess må vente lenge, er ikke ideelt og vil

ikke følge de målene/prinsippene vi ønsker å oppfylle ved en scheduler. Dette ble gått igjennom i forelesningen om “Scheduler”, spesifikt når Pål diskuterer om målene/prinsipper vi burde ha ved en “Scheduler”. Derfor er ikke SFJ ideelt med tanke på at sekvens “C” må vente lenge i denne konteksten og den faste prioriteten som “A” og “B” har.

**Hvorfor velge SJF?**
Hvis målet er å få utført flest mulig prosesser på kort tid uten hensyn av rettferdighet om at alle skal få lov til å kjøre sin prosess, så er denne algoritmen aktuell å anvende. Altså størst “Throughput”, som er et annet ord for den overordnede setningen over.

**Hvilke ytelseskriterier bør anvendes på skedulereren for et slikt operativsystem?**

c) For denne typen operativsystem er det viktig at scheduleren tar hensyn til at både CPU-bundne og I/O-bundne prosesser kan kjøres om hverandre, slik at ressursene brukes på best mulig måte. Om en kjører først alle prosessene som er CPU-bundne og så alle som I/O-bundne eller omvendt, vil ressursene ikke utnyttes på en god og effektiv måte, da den ene blir stående over lengre tid uten å gjøre noe. Dette bør unngås og en bør ha en balansert arbeidskraft på både CPU og I/O.

I tillegg er det viktig at et slikt system er rettferdig. Dette innebærer at alle prosesser får tid til å kjøres og at venting ikke blir for lang. For eksempel vil SJF algoritmen i dette tilfellet ikke være ideell da prosess C alltid vil være kortere enn både prosess A og B, og kan derfor risikere å ikke få kjørt i det hele. Da vil kanskje prioriteringssystem eller oppdeling i time-slots være mer rettferdig.

Til denne typen operativsystem vil det holde at scheduleren er statisk, da prosessene skjer med forutsigbare, jevne mellomrom i tid, og kan dermed enkelt estimeres/planlegges. I tillegg vil det, siden det ikke forventes å oppstå noen uventede prosesser som skal kjøres før andre i køen, være hensiktsmessig å ha en non-preemtive scheduler. Det innebærer at alle prosesser vil få kjøre ferdig, uten at noen bryter inn, samt det vil gi færre bytter/brytninger som gjør at det blir mindre overhead.

Besvarelsen over, er ganske bra mente Pål. Som nevnt i forelesningen om scheduler i forhold til dette med prinsipper og mål, så var det slik at: Ingen prosess burde vente for lenge. Dermed så burde ikke en SJF være ideell i dette tilfellet ettersom prosess “C” må vente lenge med å utføre sin prosess på grunn “A” og “B”.

Dermed kan vi også forkaste tanken om å benytte av en preemptiv-algoritme, fordi hvis vi skal basere prosessene etter prioritet, så vil prosess “C” måtte vente lenge til å utføre sin prosess. Scheduleren burde fordele tiden det tar for prosessene å utføre sin arbeidsoppgave rettferdig. Derfor burde en non-preemptiv-algoritme være et godt alternativ i denne sammenhengen.

I oppgaveteksten så har vi derimot ikke en beskrivelse over hvilke prosesser som er I/O-bundne eller CPU-bundne, men vi kan gjøre noen antakelser og påstå at både “A” og “B”

er I/O og “C” er CPU. Hvis dette er tilfellet, så må scheduleren rettferdig fordele tiden for disse prosessene. Årsaken er at det ikke er ideelt med tanke på ressursbruken hvis vi gjør alle CPU-bundne prosesser først, og deretter I/O-bundne prosesser eller motsatt. Da vil ikke ressursene utnyttes på en effektiv og god måte for som vi ser på bildet vi er blitt utgitt, så betyr det at ved venting, så vil CPU´en ikke ha noen klare prosesser som kan utføre sitt arbeid. Da henviser jeg til første spørsmål til “Oppgave 2” i forbindelse med prosessor “C” hvor den har en lang ventetid etter at den har fått kjørt sin prosess. En effektiv tankegang ville vært å la både “A” eller “B” få kjørt rett etter at “C” er ferdig for å bedre utnytte CPU´en effektivt.

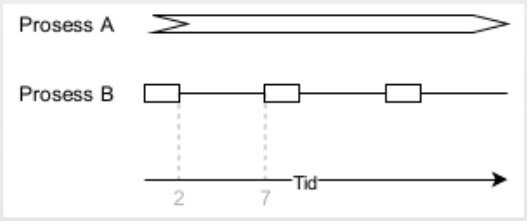
## ***Oppgave 3: Round Robin***

### 3. *Round Robin*

**Norsk**

Interaktive programmer krever en form for preemtiv skedulerer for å tilby god ytelse. Som et hobbyprosjekt jobber du med et operativsystem for arbeidsstasjoner. For å gjøre det enkelt har du valgt skeduleringsalgoritmen Round Robin. Figur 2 viser utbruddsmønstrene for to typiske prosesser som kan kjøre på systemet: én CPU-bunden og én I/O-bunden. Tiden mellom utbruddene brukes på diskoperasjoner.

- Hvordan vil lengden på tidsintervallet påvirke diskutnyttelse? Vurder intervaller på henholdsvis 50 ms og 1 ms. Det later til at kortere intervaller sørger for bedre diskutnyttelse. Hvorfor ikke gjøre intervallene så korte som mulig?
- Hvilke andre mål bør veies mot maksimal diskutnyttelse når man velger skeduleringsalgoritme for et slikt operativsystem?




Figur 2: Utbruddsmønstre for to prosesser på en arbeidsstasjon

***Viktig å nevne at “Prosess A” er en CPU-bunden prosess imens “Prosess B” er en I/O-bunden prosess.***

**Hvordan vil lengden på tidsintervallet påvirke diskutnyttelse? Vurder intervaller på henholdsvis 50 ms og 1 ms. Det later til at kortere intervaller sørger for bedre diskutnyttelse. Hvorfor ikke gjøre intervallene så korte som mulig?**

Kortere intervaller vil vanligvis gi bedre disk-utnyttelse. Dersom man skal kjøre prosessene i 50 ms vil prossess A kjøre i 50 ms, også kjøres prosess B i 2 ms før den kaller på disken,så vil prosess a kjøres igjen på nytt i 50 ms før prosess begynner igjen. rundene blir da 50 + 2 ms lange hvor disken blir brukt i 5 ms. (antar at det ikke er kostbar å contextswitche)

Dersom man kjører prosessene i 1 ms , kommer prosess a til å kjøre i 1 ms, så prosess b 1 ms, så prosess a i 1 ms så prosess b i 1 ms så vil b kalle på disken som vil kjøre i 5 ms. da blir rundene 9 ms lange hvor disken blir utnytta i 5 ms. (antar at det ikke er kostbar å contextswitche)

Context switch er en kostbar operasjon, og dersom man må bytte prosessene ofte vil det gi stort overhead.

For å besvare på spørsmålet så burde vi først se hvor lang en runde vil tilsvare for intervallet som er oppgitt i spørsmålet. Gjennom bildet i oppgaveteksten, så ser vi at den CPU-bundne prosessen “A”, ikke vil avbrytes i noen tilfeller. Imens prosess “B”, vil måtte avbryte ettersom det eren I/O-bunden prosess hvor disken vil bli utnyttet etter at “B” har kjørt sin prosess. Før vi utfører dette så antar vi at “context-switching” ikke er så kostbart. Om vi ser på tallene vi har blitt oppgitt fra oppgavebeskrivelsen, så vil prosess “A” kjøre i 50 ms. Deretter vil prosess “B” kjøre i 2 ms, hvor disken vil bli utnyttet etter 2 ms. Disken vil da bli utnyttet i 5 ms. En runde vil da tilsvare 50+2 ms hvor disken blir brukt i 5 ms. Vi vil få 9,6 % diskutnyttelse og kan vises gjennom følgende regning: (5ms(disken)/50ms + 2ms) = 9,6%

Dersom vi benytter av en kortere intervall hvor vi lar begge prosessene kjøre i 1 ms f.eks, så vil følgende skje. Prosess “A” kjører i 1 ms, videre prosess “B” i 1 ms, deretter prosess “A” i 1 ms, videre prosess “B” i 1 ms og tilslutt vil disken bli utnyttet i 5 ms. Da vil rundene bli 9 ms(4 ms for både prosess “A” og “B” og 5 ms for disk-utnyttelse). Dette viser at disken vil bli utnyttet i flere tilfeller sammenlignet med det som er skrevet i spørsmålet. Vi vil få 56% diskutnyttelse som kan vises gjennom følgende regning: (5ms/(7ms+2ms)). Ved å sammenligne dette med disk utnyttelsen over, så ser vi at dette alternativet er vesentlig bedre for disk utnyttelse. Men det er viktig å nevne om “Context-switching”. Hvis dette blir inkludert og er noe som må tas hensyn til, så vil denne løsningen bli kostbar og føre til et stort overhead ettersom vi veksler mellom prosessene på grunn av den korte tidsintervallet. Konsekvensene kan være at programmet blir ganske tregt eller noe lignende. Men siden det ikke står noe om dette i oppgaveteksten så kan vi anta at “Context-switching” ikke er så kostbart.

**Hvilke andre mål bør veies mot maksimal diskutnyttelse når man velger skeduleringsalgoritme for et slikt operativsystem?**

Siden det skal være et OS for arbeidsstasjoner, med interaktive programmer, så er det viktig at prosessene ikke må vente i en evighet, slik at det blir opplevd som mye venting/heng før en bruker får startet på sin neste operasjon. Et mål som bør veies mot maksimal diskutnyttelse er overhead av å bytte mellom prosesser, om vi har 1ms intervall for eksempel og mange små jobber som prosess B, så vil det bli mye overhead av å bytte mellom hver prosess hvert 1ms - og det kan da være fordelaktig å ha en noe høyere intervall.