Hjemmeeksamen i IN2140 - Introduksjon til operativsystemer og datakommunikasjon, Vår 2020

Kandidatnummer: 15401

Oppgave 1: Function calls vs System calls

- Beskriv hvordan funksjonskall fungerer.

Funksjonskall fungerer ved at det kalles på en funksjon, bestående av funksjonsnavn, argumenter, lokale variabler, returadresse og returverdi, som deretter oppretter en stack frame med minne på stacken. Den nyeste stack frame-en blir lagt øverst på stacken og funksjonen returnerer en peker til plasseringen på stacken. 1.

- Beskriv hvordan systemkall fungerer.

Eksempler på systemkall er open, read, write, socket osv.
Systemkall fungerer ved at userspace-applikasjonen setter opp argumentene for systemkallet, for deretter å eksekvere systemkall-instruksjonene. Instruksjonene fører til en 'exception', eller en hendelse som får programmet til å hoppe til en ny adresse og utføre kode der.

- Forklar hvordan de er forskjellige.

Forskjellen mellom funksjonskall og systemkall er at systemkall har privilegier til å operere i kjernen av CPUen og kan ta i bruk tjenester integrert i systemet, mens funksjonskall er implementert i programmet og kun opererer på kode i userspace-området.

I tillegg er funksjonskall portable og kan forflyttes, noe systemkall ikke er, og systemkall er tregere enn funksjonskall på grunn av context switch, som er prosessen som skjer når CPUen bytter fra en prosess til en annen.

Oppgave 2: Process and Threads

- Beskriv hva en prosess er.

En prosess er kjøringen av et program, altså et program som har blitt lastet inn i minnet sammen med ressursene det trenger til å operere på maskinen. Hvert program/applikasjon har som regel kun én prosess tilknyttet seg, selv om det ikke alltid er tilfellet. Noen av de mest essensielle ressursene er blant annet registre, som holder på nødvendige data, programteller/instruksjonspeker, som inneholder adressen til instruksjonen som til enhver tid blir utført, og en stack, en datastruktur hvor de aktive subrutinene, eller sekvenser av instruksjoner, blir lagret. Operativsystemet bidrar blant annet med å håndtere disse ressursene. 2.

En prosess kan befinne seg i ulike tilstander. For eksempel kan tilstanden være *running*, hvor prosessen kjører på CPUen, *blocked*, hvor den venter på at en ekstern hendelse skal inntreffe, eller *ready*, hvor prosessen er klar til å kjøre, men venter på tillatelse, eller på å

bli schedulert. I tillegg har man tilstandene *process creation* og *process termination*, hvor en prosess blir hhv opprettet eller terminert.

Operativsystemet trenger blant annet å kjenne til tilstanden til prosessene, være i stand til å opprette nye prosesser, veksle mellom dem, såkalt *context switch*, pause prosesser, terminere dem osv. til riktig tid, for å kunne oppnå en høy effektivitet og produktivitet. Dette gjør operativsystemet ved hjelp av en Process Control Block (PCB) for hver prosess, som inneholder informasjon om prosessen. På denne måten kan operativsystemet håndtere ulike prosesser på en effektiv måte gjennom å lagre tilstanden til prosessen i PCB, bytte mellom ulike prosesser, og laste opp igjen en tidligere prosess. 3.

- Beskriv hva en tråd er.

En tråd er en mindre del av en prosess. En prosess kan ha en eller flere tråder, og kalles da hhv *single-threaded process* eller *multithreaded process*. Tråd er den minste kjørende enheten som operativsystemet kan tildele prosessortid til.
En tråd kalles også en light-weight process, fordi de har hver sin stack, men samtidig deler minne og adresse i minnet med prosessen de er en del av.

- Hvilke ressurser må byttes mellom prosesser, men ikke for tråder?

Ved context switch mellom to tråder i samme prosess må alle registre, pekere til stacken og programtellere byttes ut. Ved context switch mellom tråder i ulike prosesser må i tillegg adresseområdet for det virtuelle minnet byttes ut. 4.

- Er forskjellen mellom prosess og tråd viktig for planlegging på en server med mange brukere og mange prosesser som login.ifi.uio.no? Argumenter for ditt svar.

Ja, hvis man bruker færre prosesser og flere tråder risikerer man i større grad at det får konsekvenser for de andre trådene i prosessen dersom en av trådene får problemer. Man prioriterer da hurtighet fremfor stabilitet i programmet. I tillegg tar det vanligvis opp mindre minne å bruke flere tråder, på grunn av lavere kostnad ved context switch mellom tråder enn mellom prosesser.

Hvis man har flere prosesser og færre tråder, beskytter det i større grad mot feil i programmet og man får bedre stabilitet. Det kan også være lettere å få bedre utnyttelse av minnet med flere prosesser over tråder, fordi man lettere kan skille mellom brukt og ubrukt minne, og lettere klarere det. Så hva som er mest lønnsomt avhenger av flere faktorer. 4.

Oppgave 3: Scheduling

- Forklar preemptiv og ikke-preemptiv schedulering.

Preemptiv schedulering vil si at en prosess kan bli preempted/avbrutt, av prosesser med høyere prioritet. Prosessen fortsetter senere i samme tilstand som den ble avbrutt i. Preemptive har mer overhead, det vil si minne og tid til overs, som følge av hyppigere context switching.

Ikke-preemptiv schedulering betyr at den kjørende prosessen får gjøre seg ferdig med tidsbolken som kjører, og at prosesser med høyere prioritet må vente. Denne typen schedulering kan være nyttig for kortere oppgaver, som for eksempel å sende en pakke, og er mye brukt i blant annet disk og nettverkskort. Ikke-preemptiv schedulering gir mindre hyppighet av switcher og gir derfor også mindre overhead.

- Drøft om preemptiv eller ikke-preemptiv schedulering er passende for dette systemet.

Jeg vil tro at en ikke-preemptiv schedulering passer for systemet, fordi jeg går ut i fra at hver tråd i applikasjonen ikke skifter prosess i tilfellene der en feil i systemet blir oppdaget, og det sendes en advarsel til en annen maskin. Da vil hver prosess gjøre seg ferdig med kjørende prosess, før den venter på å motta en ny oppgave.

Dersom prosessen ikke blir avbrutt kan man også bruke en preemptive scheduleringsalgoritme, men ettersom alle prosessene har samme prioritet vil ikke prosessene blir avbrutt av en med høyere prioritet. Preemptive er scheduleringsalgoritmen som blir mest brukt i dag.

- Forklar for 4 scheduleringsalgoritmer hvordan de fungerer og diskuter hvor godt hver av dem er egnet for denne oppgaven.

FIFO (First In First Out) fungerer ved at prosessene blir håndtert i den rekkefølgen de kommer inn, og de andre prosessene venter til foregående prosess er ferdig. Denne formen for schedulering gir en rettferdig service til alle prosessene, og alle prosessene kommer i «riktig» rekkefølge, noe som kan være en fordel i for eksempel spill. Fordelen er at den er enkel, og ulempen at den har lange ventetider/time-slices.

I denne oppgaven er det en fordel at prosessene blir behandlet etter hvert som de kommer inn, for å finne ut om et system har feil så raskt som mulig. Denne scheduleringsalgoritmen kan derfor være en fordel, også fordi oppgavene ikke har noen prioritetskriterier.

Round Robin fungerer ved at oppgavene/prosessene starter med for eksempel ett sekunds tidsintervall. På denne måten blir alle prosessene ferdig omtrent samtidig som ved FIFO, men alle oppgavene får kjøre tidligere, i kortere time-slices.
Round Robin passer bedre for programmer som krever interaktivitet, enn FIFO.

RR vil bli ferdig på ca. samme tid som FIFO, men det vil være mer kostbart for CPUen å «switche» mellom de ulike prosessene, og interaktivitet er ikke like viktig for dette

operativsystemet. RR er av den grunn kanskje ikke det beste alternativet for dette systemet.

Shortest Job First er en algoritme som prioriterer oppgaver/prosesser som har den korteste prosesseringstiden, altså de oppgavene som blir ferdig først. SJF er derfor den scheduleringsalgoritmen med lavest gjennomsnittlig ventetid av alle algoritmene. En ulempe med SJF er at man må vite hvor lang prosesseringstid hver av oppgavene har i forkant, noe som ikke alltid er tilfellet. I tillegg kan det oppstå «starvation» hvis mange korte prosesser ankommer samtidig, hvor noen prosesser blir stående lenge å vente, som også gir veldig lang «turnaround time», som vil si tiden det tar for enkelte prosesser å behandles og ferdigstilles. 5.

Denne algoritmen kan være effektiv med mange korte jobber, men den er kanskje ikke den rette fordi den baserer seg på prioritering av oppgaver. I tillegg kan den føre til at noen oppgaver ikke får kjørt, som kan hindre at det oppdages feil.

Lottery er en scheduleringsalgoritme hvor prosessene blir schedulert i tilfeldig rekkefølge. Scheduleringen går ut på at hver prosess får tildelt ett «lodd» for så at scheduleren velger ut et tilfeldig lodd og kjører prosessen med dette loddet. Prosesser med et høyere antall lodd vil da ha en større sjanse for å bli valgt, og dermed kan man manipulere hvilke prosesser som får høyest prioritet. Prosessene som får flest lodd er de som forventes å ta kortere tid, og derfor ligner denne algoritmen litt på SJF, bortsett fra at den er mer rettferdig og man unngår «starvation» ved at alle prosessene får en sjanse.
En ulempe ved lottery er at det er uforutsigbart og potensielt kan gi en ubalanse i CPU-tid mellom prosessene, og derfor er den ikke mye i bruk i dag. 6.

Denne algoritmen kan gi for mye usikkerhet til oppgaven, ettersom det er viktig at prosessene får noenlunde lik CPU-tid, sånn at de prosessene som kommer til å oppdage feil i systemet ikke risikerer å bli nedprioritert.

- Foreslå en av disse 4 scheduleringsalgoritmene og forklar valget ditt.

Jeg ville valgt FIFO til dette systemet fordi ingen av oppgavene har prioritet og fordi det innebærer at prosessene blir behandlet etter hvert som de kommer inn, noe som gir kortest tid til å avdekke feil i systemene. I tillegg er det enkelt å implementere og alle prosessene blir behandlet likt, slik at man unngår «starvation» ved at noen prosesser ikke blir kjørt, og man unngår også lang «turnaround time», som gjør at det kan ta lang for noen av prosessene før de er ferdig.

Oppgave 4: Virtual memory

Virtuelt minne er en «mapping» mellom adressen programmet bruker og adressen i minnet, såkalte page-tables. Motivasjonen bak virtuelt minne var at RAM tidligere var dyrt og det var behov for å kjøre programmer på PCen selv når den gikk tom for RAM og effektivisere bruken av RAM. Virtuelt minne gjør det mulig å bruke RAM-minnet mer effektivt og fleksibelt.

- Forklar (generelt sett) hvordan virtuelle adresser blir oversatt til fysiske adresser med multi-level paging. Forklaringen må inkludere problemer som minneoppslag, page cache, innlasting av sider og begrepet «dirty page». Det er ikke nødvendig å forklare strategier for utskiftning av sider.

Virtuelle adresser blir oversatt til fysiske adresser med multi-level paging ved at man først finner og aksesserer den første page-tabellen ved å oversette de 10 første bitsene av den virtuelle adressen. Deretter oversettes de 10 neste bitsene i adressen til page-tabell nummer to, der den fysiske adressen til dataene vi ønsker å aksessere ligger. Disse page-tabellene må til enhver tid ligge i minnet, altså en første-nivå page-tabell, slik at vi kan finne de andre page-tabellene, og minst en annen-nivå page-tabell som gir oss den fysiske adressen. I tillegg til de to page-tabellene har vi offset, som adresseres av resten av bits-ene i den virtuelle adressen.

Et problem som kan oppstå er hvis to page-tabeller forsøker å aksessere det samme minneområdet, noe som kan føre til at programmet krasjer. Virtuelt minne tilbyr en løsning på problemet ved at alle programmene har det samme fysiske minneområdet, men får tildelt hvert sitt virtuelle minneområde, som igjen blir «mappet» til det fysiske minnet i RAM.

For å gjøre det virtuelle minnet raskest mulig bruker man en page table cache som kalles Translation Lookaside Buffer (TLB). TLB ligger mellom prosessoren og RAM, og gjør det mulig å laste inn page-tabellen raskt dersom den ligger i TLB. Hvis page-tabellen ikke ligger i TLB, men i RAM, må den hentes fra RAM til TLB, og det tar lengre tid å aksessere. Dersom den ligger på disk tar det enda lengre tid. Løsningen er å utvide TLB til å dekke så store mengder data som mulig, bruke flere TLB, eller å ha hardware som automatisk fyller TLB.

Ved innlasting av pages gjør multi-level paging at man blant annet kan laste inn prosesser med større adresseområde enn det man i utgangspunktet har plass til, og det sparer tid ved context switch.

Hvis en page er markert som «dirty», det vil si at den har blitt modifisert siden vinduet med page-er ble lastet inn, betyr det at innholdet i minnet og på disk er forskjellig, og dersom page-vinduet blir byttet ut, må innholdet kopieres til disk av en «page fault handler». 7. 8.

- Hvor mange nivåer må din virtuelle adresseoversettelse minst ha med den oppgitte sidestørrelsen? Forklar hvorfor. Ikke glem at page tables og page maps også er begrenset av 1 kilobyte sidestørrelse.

Oppgave 5: Splitting an address block into networks

- Hva er nettmasken til adresseblokken som er tildelt selskapet ditt?

Nettverksmasken til adressen 9.239.16.0 er 255.255.252.0, siden adresseblokken består av 1024 IPv4-adresser. Standard nettverksmaske for adresser der første oktett er mellom 0-127 er 255.0.0.0.

- Hvor mange adresser tildeler du hvert nettverk, netA, netB og netC? Hva er nettverksadressen og nettmasken for hvert av disse nettverkene? Forklar hvordan du kommer til denne løsningen.

For å finne tre subnett bruker jeg nettverksmasken 255.255.192.0 eller 11111111.11111111.11000000.00000000.
Dette fordi jeg må «ta» to 1-bits fra nettverksmasken for å lage 4 nettverk, som er det jeg minst har behov for. CIDR-notasjon blir da 9.239.16.0 /18.
Dette gir 2^14 (antallet 0-bits igjen i nettverksmasken) = 16 384 host per nettverk og 16 382 mulige hosts, fordi en adresse settes av til nettverksadressen og en adresse er broadcast-adressen.
For å finne «rangen» av IP-adresser i hvert nettverk trekker jeg 192 i nettverksmasken fra 256 og får 64. Subnett-«rangen» blir da 0-63. 9.

Nettverk 1: Nettverksadresse: 9.239.0.0 Range: 9.239.0.1 – 9.239.63.254

Subnett1/netA: Nettverksadresse: 9.239.64.0. Range: 9.239.64.1 – 9.239.127.254

Subnett2/netB: Nettverksadresse: 9.239.128.0 Range: 9.239.128.1 – 9.239.191.254

Subnett3/netC: Nettverksadresse 9.239.192.0 Range: 9.239.192.1 – 9.239.255.254

- Forklar hvorfor du har valgt å dele adresseområdet ditt i tre nettverk som dette. Vurderte du fremtidig vekst i oppgaven din? Beholdt du noen adresser i reserven? Og hvis ja, hvordan planlegger du å bruke dem?

Jeg har valgt å dele adresseområdet i tre nettverk på denne måten, fordi det er den generelle metoden for å dele inn i subnettverk. Det mest ideelle hadde kanskje vært å ha færre hosts, men da må man vel dele inn nettverkene mer manuelt, ved å for eksempel ha et intervall på 80:

Subnett1/netA: Nettverksadresse: 9.239.16.0
Nettverksmaske: 255.255.255.0
Range: 9.239.16.1-79 /24
Antall hosts: 254

Subnett2/netB: Nettverksadresse: 9.239.16.80
Nettverksmaske: 255.255.255.0
Range: 9.239.80-127 /24
Antall hosts: 254

Subnett3/netC: Nettverksadresse: 9.239.16.240
Nettverksmaske: 255.255.255.0

Range: 9.239.128-239 /24
Antall hosts: 254

Mulig at dette er bedre enn den opprinnelige løsningen, men jeg valgte å ta med begge.
Med 254 hosts per nettverk har man da ekstra rom for å utvide, og benytter en stor andel av de adressene man har fått utdelt, sammenlignet med for eksempel 128 hosts per nettverk. De resterende IP-adressene kan brukes ved vekst og/eller til ekstra enheter, som skrivere, rutere osv.

Oppgave 6: Packets over TCP

- Forklar hva uttalelsen «TCP leverer en byte-stream-tjeneste» betyr.

Uttalelsen «TCP leverer en byte-stream-tjeneste» betyr at TCP-protokollen leverer en tjeneste som tilbyr en «strøm» av bytes levert over nettverket i riktig rekkefølge gjennom en sikker, etablert tilkobling.

- Hva betyr byte-stream-tjenesten for applikasjoner som bruker socket-funksjoner (Berkeley-sockets) som send() og recv() for å utveksle data? Hva må programmerere gjøre annerledes når de bruker TCP i stedet for en transportprotokoll som gir pakkebasert tjeneste?

I applikasjoner som benytter seg av byte-stream-tjenester og bruker socket-funksjoner som send() og recv() for å utveksle data, er det nødvendig å opprette en kobling mellom klient og server ved sending, i motsetning til for eksempel UDP som benytter en forbindelsesløs tjeneste. Programmereren må derfor benytte funksjoner som connect(), write() og read() i klienten og listen() og accept() i server(), hvor det opprettes en sikker kobling mellom klient og server, i stedet for sendto() og recvfrom(), som brukes for UDP.

Programmereren må håndtere meldingsgrenser som ikke finnes når dataene blir sendt som en byte-strøm, uten pakker, og, h*n må håndtere latens som oppstår når pakker går tapt, i og med at pakkene blir sendt i rekkefølge, og de pakkene som kommer etter en tapt pakke blir stående i kø mens pakken sendes på nytt.

I tillegg kan det tas høyde for overhead på 3-way handshake i starten av TCP, hvor man kan vurdere å gjenbruke socketen hvis man skal sende mye data.

- Hvis en applikasjon bruker TCP men trenger pakker, hvordan må utvikleren designe applikasjonslagsprotokollen?

Utvikleren må designe applikasjonslagsprotokollen med meldingsgrenser dersom TCP trenger pakker.

- Er det enkelt å lage en ny transportprotokoll PacketTP med små endringer i TCP, men som leverer en forbindelsesorientert, pålitelig, ende-til-ende datagramtjeneste? Foreslå minst 3 endinger som er nødvendige og forklar hvorfor de er viktige. Endringene kan være små.

Ja, man må definere blant annet kommandoer og payload.

Oppgave 7: Flow control

- Forklar flytkontrollmekanismen «selective repeat».

Selective repeat-protokollen gjør det mulig å ta imot pakker som kommer etter ødelagte eller tapte pakker, ved å tillate mottakeren å akseptere pakker som ikke kommer i rekkefølge.

- Forklar flytkontrollen kreditt-mekanisme.

Flytkontrollen kreditt-mekanisme fungerer ved at man har oppsamlinger av data i buffere underveis, for å hindre at strømmen av data stopper ved flaskehalser. Den foregår ved at sender først etterspør nødvendig buffer. Deretter setter mottakeren av buffer og returnerer ACK som bekreftelse i tillegg til buffer-kreditt/tildelt buffer. Når all kreditten er brukt opp blir senderen blokkert.

- Beskriv forskjellene mellom selective repeat og kreditt-mekanisme. Er det noen forskjell ved håndteringen av pakketap, tap av ACKer eller bruk av timeouts? Hvor stort kan sekvensnummerområdet være, og hvor mange av disse sekvensnumrene kan være «in flight» til et gitt tidspunkt?

Selective repeat håndterer pakker ved å sende savnede pakker eller pakker med feil på nytt. Når det gjelder ACK-er benytter den ekstra bits i headeren for å håndtere dette, og den venter på timeout for hver pakke separat, i stedet for å anta at alle pakkene er tapt dersom en er tapt.

Kreditt-mekanisme håndterer pakketap ved å vente til timeout før den sender pakken på nytt, og håndterer tap av ACK-er ved å vente.

- Kan du finne et problem med tap av ACK-er i kreditt-mekanismen som ikke eksisterer i selective repeat, og har du et løsningsforslag?

Den kan bli stående å vente lenge. 10.

Oppgave 8: Building Routing tables from Dijkstra Shortest Path First

- Forklar hvordan Dijkstras shortest path first algorithm fungerer.
  Hvorfor er disse trinnene viktige? Hva gjør de?

Djikstras shortest path first algorithm gjør en i stand til å finne den korteste ruten mellom en startnode og hvilken som helst annen node i nettverket.
Algoritmen baserer seg på fire trinn:

1. Node A blir merket som «permanent» med en fylt sirkel. I utgangspunktet er ingen ruter kjent og alle andre noder er markert med uendelighetstegn. Node A er startnoden.
2. Merk alle direkte tilstøtende noder med avstanden til A. Alle noder som ligger som nabo til A blir merket.
3. Undersøk alle ikke-permanente noder og gjør noden med den minste «labelen» permanent. Nodene som nettopp ble merket blir sammenlignet, og den noden med kortest avstand til arbeidsnoden er den nye «permanente» noden.
4. Denne noden vil være den nye arbeidsnoden for den iterative prosedyren. Fortsett med steg 2. Man bruker den nye permanente noden videre og fortsetter fra steg 2. 11.

- Bruk algoritmen til å beregne avstanden fra A til alle andre noder. Vis 3 mellomtrinn: rett etter at F har fått en permanent label, rett etter at E har fått en permanent label, rett etter at I har fått en permanent label.

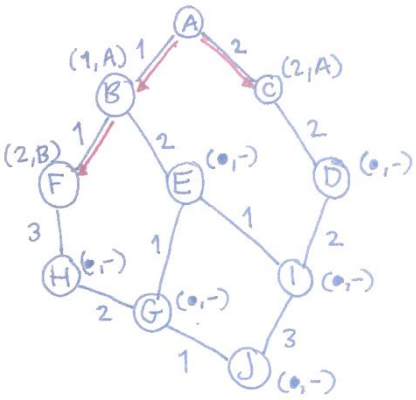
Beregner avstanden fra A til alle noder (avstanden er tallet på nederste rad i tabellen):

V	A	B	C	D	E	F	G	H	I	J
A	0_A	1_A	2_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	2_A	∞_A	3_B	2_B	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	2_A	4_C	3_B	2_B	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	2_A	4_C	3_B	2_B	∞_A	5_F	∞_A	∞_A
E	0_A	1_A	2_A	4_C	3_B	2_B	4_E	5_F	4_E	∞_A
D	0_A	1_A	2_A	4_C	3_B	2_B	4_E	5_F	4_E	∞_A
G	0_A	1_A	2_A	4_C	3_B	2_B	4_E	5_F	4_E	5_G
I	0_A	1_A	2_A	4_C	3_B	2_B	4_E	5_F	4_E	5_G
H	0_A	1_A	2_A	4_C	3_B	2_B	4_E	5_F	4_E	5_G

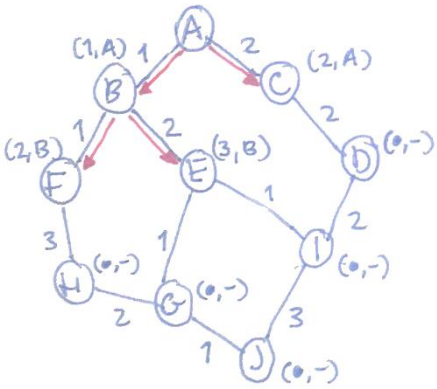
A B E G J

A B E G J er den korteste ruten ut i fra tabellen. J sin permanente node går til G, G sin permanente node går til E osv. 12.

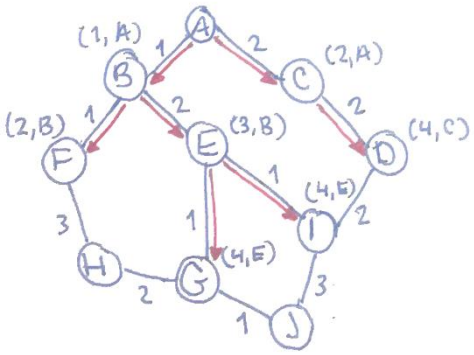
Etter at F har fått en permanent label:



Etter at E har fått en permanent label:



Etter at I har fått en permanent label:



- Til slutt lager du en tabell som viser A sin routingtabell, bestående av tre kolonner. Første kolonne: destinasjonsnode, andre kolonne: neste node på stien til destinasjonen, tredje kolonne: lengden på den korteste stien til denne.

Destinasjonsnode	Neste node	Lengde på korteste sti
B	A	1
E	B	2
G	E	1
J	G	1
		= 5

(Ikke helt sikker på om jeg har forstått denne deloppgaven riktig).

Kildeliste:

1. https://zhu45.org/posts/2017/Jul/30/understanding-how-function-call-works/
2. https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/
3. https://medium.com/@imdadahad/a-quick-introduction-to-processes-in-computer-science-271f01c780da
4. https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/
5. https://www.guru99.com/shortest-job-first-sjf-scheduling.html
6. https://intro2operatingsystems.wordpress.com/tag/lottery-scheduling/
7. https://www.youtube.com/watch?v=Z4kSOv49GNc&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=12
8. http://www.cs.rpi.edu/academics/courses/fall04/os/c12/index.html
9. https://searchnetworking.techtarget.com/tip/IP-addressing-and-subnetting-Calculate-a-subnet-mask-using-the-hosts-formula
10. https://www.uio.no/studier/emner/matnat/ifi/IN2140/v20/forelesningsvideoer/04-04-flowctrl-part-2.mp4
11. https://www.uio.no/studier/emner/matnat/ifi/IN2140/v20/slides/07-02-route-dijkstra.pdf
12. https://www.youtube.com/watch?v=0nVYi3o161A

+ forelesningsnotater.