Hjemmeeksamen - INF1060 - høst 2016

I denne oppgaven skal du bruke det du har lært til nå og gjort i oblig 1 og 2, og kombinere det med nettverksprogrammering og IPC (Inter Process Communication). Vi skal med andre ord jobbe med filer, structer, strenger, pekere, systemkall, nettverk, IPC og mer!

Oppgavene skal løses **individuelt**, se ellers forskrift om studier og eksamen. Du vil finne nyttige ukesoppgaver med tilhørende forklaringer av viktige begreper på gruppelærersiden, og som i de fleste fag er semestersiden stedet for forelesningsfoiler og annen informasjon du kanskje er på utkikk etter. Du kan også finne noen relevante eksempelprogrammer på INF1060-githuben.

Dersom du har spørsmål underveis kan du oppsøke en orakeltime eller stille spørsmål på Piazza. Det vil (selvfølgelig) være mye relevant informasjon i plenumstime.

Husk å lese hele oppgaveteksten, så du vet hvor mye arbeid du har igjen totalt sett. *Du vil til slutt i dokumentet finne en liste som viser hva du bør prioritere når du jobber med oppgaven.*

Oppgavene blir testet på Linux på Ifi sine maskiner eller tilsvarende.

Innlevering i Devilry innen fredag 18.11 23:59. Denne tidsfristen er **hard**, leveringer etter dette blir vurdert som **F**.

Denne hjemmeeksamenen teller 40% av endelig karakter i INF1060 og må bestås for å kunne få endelig karakter.

Oppgaven

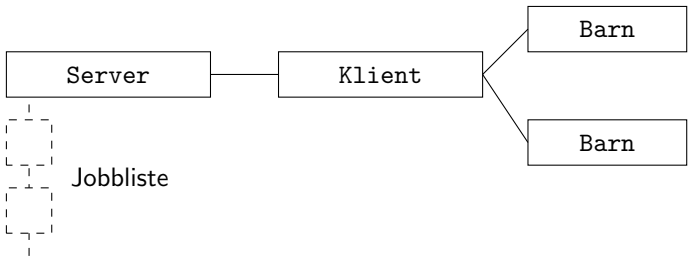
I denne oppgaven skal du programmere en klient og en server som skal snakke med hverandre. Serveren har en del jobber den vil ha utført, og klienten skal utføre disse jobbene. På server-siden ligger det en liste med jobber i en joblist-fil, og jobbene i den skal oversendes til klienten for utførelse. Serveren og klienten skal kommunisere over **TCP**. Forholdet mellom prosessene er illustrert i figur 1.

Programmene skal kompileres med make (**du må altså lage en makefile**).

Du finner flere joblist-filer du kan teste programmet ditt med i Hjemmeeksamen-repositoriet på github

Spesifikasjoner

Her er litt mer spesifikk informasjon om hvordan programmene skal fungere, inndelt i en seksjon om klienten og en om serveren.



Figur 1: Arkitektur

Klient

Klienten skal starte to barneprosesser med `fork` for å utføre jobber. Klienten og barneprosessene dens skal kommunisere med *pipes*. Klienten skal skrive i piper til barneprosessene, en pipe til hver barneprosess. Klienten skal deretter koble seg på serveren med kall på `socket` og `connect`. Barneprosessene skal altså ikke være koblet til serveren.

Etter at pipene er satt opp, barneprosessene er forket ut, og klienten har koblet seg på serveren, skal klienten gi brukeren fire valgmuligheter:

- Hent én jobb fra serveren
- Hent X antall jobber fra serveren (spør bruker om antall)
- Hent alle jobber fra serveren
- Avslutte programmet

For å hente jobber fra serveren skal klienten sende get-job meldinger (definert under) til serveren.

Kommunikasjon med server

Klienten kommuniserer med serveren med meldinger bestående av én byte, der bytens innhold har følgende betydning:

- `'G'` - En `get-job`-melding
- `'T'` - En melding som varsler serveren om at klienten terminerer normalt.
- `'E'` - En melding som varsler serveren om at klienten terminerer på grunn av en feil.

I utgangspunktet sender klienten én get-job-melding per jobb den henter fra serveren, men i tilfeller der brukeren ber om et antall jobber, eller alle jobber, går det an å løse det bedre enn å sende én get-job-melding per jobb. Hvis du implementerer en annen løsning for disse tilfellene må du dokumentere det godt.

Jobbtyper

Jobbtype er definert som en `char`. Klienten skal lese jobbtype-feltet i meldingen fra serveren og sende jobbteksten til en av sine to barneprosesser.

Jobbtype	Sendes til barneprosess nr	Jobbbeskrivelse
`'O'`	1	Jobbteksten skal printes til standard out (`stdout`).
`'E'`	2	Jobbteksten skal printes til standard error (`stderr`).
`'Q'`	1 og 2	Barneprosessene skal terminere

Barneprosessene 1 og 2 har altså bare én oppgave hver, å skrive jobbteksten til `stdout` og `stderr`.

Figur 2: En jobb-melding sendt fra serveren

Kjøring

Startes med følgende argumenter:

- Adresse - (IP-) adressen der serveren kjører
- Port - Porten serveren lytter på

Serveren må være startet først på den angitte adressen med samme port-nummer for at klienten skal lykkes med å koble seg opp. For eksempel:

```
$> ./klient 127.0.0.1 4323
```

Du kan legge til ytterligere argumenter om du trenger det, men sørg for å dokumentere det så rettere vet hvordan de skal starte programmet. **Bonus:** gjør det mulig å koble seg opp med maskinnavn i stedet for IP-adresse, f.eks. `vor.ifi.uio.no`.

Server

Serveren skal åpne en joblist-fil for lesing (se egen seksjon om filstruktur), og sette opp en *socket* ved hjelp av kall på `socket`, `bind` og `listen`, og akspetere tilkobling fra klienten med `accept`. Serveren skal kun håndtere én klient, og avvise tilkoblingsforsøk fra andre enn den første klienten.

Når serveren mottar en melding fra klienten som spør om en ny jobb, skal serveren svare med en jobb fra joblist-filen den har åpnet. En jobb skal leses fra filen først når klienten spør om en jobb, og serveren skal da sende en melding til klienten. Altså skal *ikke* hele filen leses inn i minnet fra starten av programmet.

Feltene i meldingen som serveren skal sende til klienten, illustrert også i figur 2:

- `T` - Jobbtype, `char`
- `L` - tekstlengde, `unsigned char`
- Jobbtekst - `char[]`

Serveren må kunne håndtere alle meldinger klienten sender, som definert i klient-seksjonen.

Når det ikke er flere jobber igjen i jobliste-filen skal serveren sende en melding med jobbtype `'Q'` og tekstlengde 0, for å fortelle klienten at det ikke er flere jobber. Deretter skal serveren vente på at klienten bekrefter at den avslutter, før serveren selv avsluttes.

Filstruktur

Hver av jobbene i filen ser ut som følger:

- Jobbtype - `char`
- Tekstlengde - `unsigned char`

- Jobbtekst - `char[]`

Lengden på jobbtekst-arrayet er gitt i tekstlengde-charen. **Denne teksten er ikke nullterminert.** Jobbtype er forklart i seksjonen om klienten. Merk at jobbene ikke er adskilt med linjeskift, null-byte eller noen annen separator. Figur 2, som illustrerer en melding som serveren skal sende, er lik strukturen til en jobb i filen.

Kjøring

Serveren startes med følgende argumenter:

- Filnavn - navnet på en fil som inneholder en liste med jobber som leses og sendes til klienten på forespørsel.
- Port - porten som serveren skal lytte på.

For eksempel kan man starte serveren på denne måten:

```
$> ./server 1job.joblist 4323
```

Du kan legge til ytterligere argumenter om du trenger det, men sørg for å dokumentere det så rettere vet hvordan de skal starte programmet.

Feilhåndtering

Programmene bør ikke krasje under noen omstendighet. Returverdier fra funksjonskall bør sjekkes og feil bør håndteres på en god, oversiktlig og brukevennlig måte.

Programmene bør testes med `valgrind`. Eventuelle feil som valgrind påpeker bør utbedres.

Avlutning

Når programmene avslutter skal alle allokerte minneområder (allokert med `malloc`) frigis ved kall på `free`. Sockets skal stenges i en rekkefølge som gjør at programmene ikke krasjer eller henger. Pipene som klienten og barneprosessene bruker skal lukkes. **Obs!** Husk på å frigjøre/lukke i barneprosesser. Pass også på at du ikke får zombie-prosesser (zombie-prosesser er barneprosesser som lever etter at foreldreprosessen har avsluttes, som aldri avslutter av seg selv).

Ctrl+C (sigint)

Dette anbefales det å gjøre først når resten av programmet fungerer.

Sett opp en signal-handler som gjør at brukeren også kan trykke `Ctrl-C` for å terminere server og/eller klient, uten at dette fører til minnelekasjer eller at det andre programmet fryser/krasjer.

Viktighet av de forskjellige funksjonalitetene

Her er en liste over viktigheten av de forskjellige delene av oppgaven. Bruk listen som en guide for hva du bør jobbe med (og i hvilken rekkefølge). Dette vil bli brukt ved retting.

1. Fungerende kommunikasjon mellom server og klient.

2. Fungerende kommunikasjon mellom klient og barneprosesser.
3. God og korrekt bruk av minne (heap og stack) (`free`).
4. God programstruktur (filer, funksjoner).
5. Godt dokumentert kode

Dokumentasjon

Koden du leverer skal være godt dokumentert. Foran hver metode du lager skal det være en kommentarblokk som inneholder en beskrivelse av hva funksjonen gjør, hva input-parametrene betyr og hva funksjonen returnerer. For eksempel:

```
/* This function takes two integer arguments, adds them
 * together and multiplies the result by 2. Only works if
 * the result of the addition is 0 or above.
 *
 * Input:
 *      a: the first int
 *      b: the second int
 *
 * Return:
 * The result of the calculation, or -1 if the result of the
 * addition was negative.
 */
int add_and_double_result(int a, int b) {
    return a+b >= 0 ? (a+b) << 1 : -1;
}
```

Selv om koden inne i funksjonen er litt kryptisk er det lett å forstå hva funksjonen gjør fordi den er godt dokumentert!

Levering

1. Lag en mappe med ditt kandidatnr: `mkdir 12345`
2. Kopier alle filene som er en del av innleveringen inn i mappen:
   `cp *.c 12345/` (f.eks.)
3. Komprimer og pakk inn mappen:
   `tar -czvf 12345.tgz 12345/`
4. Logg inn på Devilry
5. Lever under INF1060 Hjemmeeksamen

Kandidatnummer finner du på studentweb.

Relevante man-sider

Disse man-sidene inneholder informasjon om funksjoner som kan være relevante for løsing av denne oppgaven. Merk at flere av man-sidene inneholder informasjon om flere funksjoner på én side, som `malloc/calloc/realloc/free`. Nummeret foran hver funksjon er hvilken seksjon i manualen siden ligger i. Får å få informasjon om f. eks. `read`, skriv `man 2 read`.

- 3 `malloc/calloc/realloc/free`
- 3 `fgets/fgetc/getchar`
- 3 `fread/fwrite`
- 3 `fopen/fclose`
- 3 `scanf/fscanf`
- 2 `read`
- 2 `write`
- 2 `socket`
- 2 `bind`
- 2 `listen`
- 2 `accept`
- 2 `connect`
- 2 `signal`
- 2 `pipe`
- 3 `strcpy`
- 3 `memcpy`
- 3 `memmove`
- 3 `atoi`
- 3 `strtol`
- 3 `isspace/isdigit/alnum` m.fl.
- 3 `strdup`
- 3 `strlen`
- 3 `printf/fprintf`