UiO : Universitetet i Oslo

Obligatorisk oppgave 2 i INF1060:

ifish - en kommandotolker for Linux

Utlevering: 14. oktober 2015
Innlevering: 28. oktober 2015, kl. 20:00
Vedlegg: 1. safefork.c

Intro

For å sende kommandoer til operativsystemet (OSet) og for OSet å gi respons tilbake bruker vi ofte et *shell* - også kalt en *kommandotolker*. Unix-systemer har ikke noe innebygget vindu-basert grensesnitt, men antar et enkelt tegnbasert grensesnitt hvor en bruker skriver inn strenger av tegn (tekst) og avslutter med ENTER eller RETURN.

Denne oppgaven går ut på å programmere et slikt shell i programmeringsspråket C for Linux á la sh, csh eller bash, men i en sterkt forenklet utgave (programmet skal kunne kjøres på IFIs standard Linux-maskiner som de på termstua eller Linux-clusteret - `ssh linux`). Det skal kunne eksekvere vanlige kommandoer med parametere samt at det skal ha noen innebygde kommandoer i likhet med de vanlige kommandotolkerene.

Oppgaven

Dette er en omfattende oppgave, og det kan være lurt å ta den stegvis. Resten av oppgaveteksten er derfor skrevet slik. Videre er en del av oppgaven å skrive ut debug-informasjon ved hjelp av *fprintf* (se `man fprintf`) til 'stderr' (ettersom en utskrift sendt til 'stderr' er ubufret, skulle man se utskrift på skjermen selv om programmet skulle kræsje. Utskrifter som går til 'stdout' kan derimot "forsvinne" i en kræsj på grunn av bufring. Se `man stderr`).

Debug utskriftene kan for eksempel gjøres slik:

```
#ifdef DEBUG
fprintf(stderr,...);
#endif
```

Denne printf-kommandoen blir bare eksekvert hvis DEBUG er definert, for eksempel ved at programmet er kompilert med `-DDEBUG` opsjonen eller å ha en `#define DEBUG` i C-filen. Det kan i tillegg til de "obligatoriske" debug-utskriftene, som er spesifisert under og som bare skrives ut i "debug modus", være lurt å legge inn mange testutskrifter underveis på samme måte for å få oversikt over hva som skjer. Sjekk spesielt om alle tekstoperasjonene fungerer.

Kompilering av koden

For å kompilere koden skal du lage en `Makefile` (se `man make`) slik at man kan skrive `make` i katalogen hvor filene er lagret for å kompilere programmet. Koden som gis ut i *safefork.c* skal ikke inkluderes i samme fil som resten av koden, men forbli slik den er utgitt - Makefilen skal passe på at den kompileres inn. Det kan også være lurt (og det er et pluss) å legge noen av de andre funksjonene (for eksempel de innebygde funksjonene) i egne filer. Makefilen skal også leveres inn og gir poeng.

Lese kommando fra tastaturet

Kommandotolkeren skal gå i evig løkke og hente kommandoer fra brukeren. Det vil si at for hver runde skal den kunne lese inn en kommando fra tastaturet. ifish skal:

- Skrive ut en ledetekst (prompt). Denne skal bestå av brukernavn@ifish hvor *brukernavn* er brukernavnet på den som er logget inn (HINT: Bruk omgivelsesvariabelen USER hvor omgivelsevariable generelt kan leses med for eksempel *getenv*.) samt et nummer (en teller) som indikerer antall kommandoer siden shellet ble startet:

  `paalh@ifish 1>`

- Lese en linje fra standard-inn (tastaturet). Vi kan anta at linjen aldri er lenger enn 120 tegn. HINT: Bruk *fgets* til å lese linjen.

- Hvis det ikke er flere linjer å lese (brukeren vil avslutte), skal ifish avslutte med statusverdi

0. NB! Dette er ikke det samme som å lese en tom linje! Når man leser fra tastaturet, kan brukeren trykke på Ctrl+D for å angi at det ikke er flere linjer å lese. HINT: Bruk funksjonsverdien som *fgets* returnerer til å sjekke om det var flere linjer å lese.

- `DEBUG:` Som debug-informasjon skal programmet skrive ut den leste linjen.

Dele opp den innleste linjen i ord - kommando med parametere

For at kommandotolkeren skal kunne eksekvere en kommando, må linjen vi leste inn over deles opp i ord skilt av blanke tegn. Det vil si at vi etter å ha lest inn en linje i oppgaven over skal kunne:

- Finne ut hvilke ord den inneholder; vi antar at det aldri er mer enn 20 ord på en linje, og et ord er definert som en sekvens ikke-blanke tegn. HINT: Bruk funksjonen '*isspace*' til å avgjøre om et tegn er blankt.

- En kopi av alle ordene skal lagres i arrayet/vektoren *char *param[21];*. Elementet param[0] skal peke på en kopi av det første ordet, param[1] på det andre ordet, osv. Elementet param[n] (der n er antall ord på linjen) skal inneholde verdien 0 (NULL) for å markere slutten på vektoren. (De oppmerksomme ser kanskje at *param* ligner på *argv* parameteren til *main*)

- `DEBUG:` Som debug-informasjon skal programmet kunne skrive ut innholdet av arrayet/vektoren *param*.

Utføre kommandoer

Fra stegene over skulle vi nå ha et program som leser input fra brukeren, deler denne strengen opp i ord og legger disse inn i vektoren *param*. Neste steg er da å få kommandotolkeren til å tolke disse ordene som kommandoer og tilhørende parametere, for deretter å utføre disse

kommandoene ved å opprette en barneprosess. Vi antar at første ord på linjen (*param[0]*) inneholder kommandonavnet og at de etterfølgende ordene er parametere:

- Hvis kommandoen er *exit* eller *quit* skal kommandotolkeren avslutte.

- Hvis linjen ikke inneholder noen ord (det vil si at den er blank), skal det ikke gjøres mer med linjen.

- Start en barneprosess med *safefork* (på grunn av faren for at genereringen av barneprosesser skal løpe løpsk, må du benytte *safefork* i stedet for *fork*. Koden til `safefork' finnes på filområdet ~inf1060/code/, på http://folk.uio.no/inf1060/programs/safefork/, eller som vedlegg under).

  Denne barneprosessen skal prøve å utføre angitte kommando med et kall på *execve*. Den skal lete blant filområdene angitt i omgivelsesvariabelen `PATH' etter hvor kommandoen kan finnes som eksekverbar fil. HINT: Omgivelsevariable kan leses med for eksempel *getenv*.

- Hvis intet program 'xyz' finnes på områdene angitt i PATH, skrives det ut en feilmelding før barneprosessen avsluttes:

  ```
  ifish: xyz: command not found
  ```

- Sjekke om linjen avsluttes med en '&' (eventuelt etterfulgt av blanke). Hvis den gjør det skal foreldre-prosessen bare skrive ut barneprosessens PID og fortsette løkken med å spørre om en ny kommando. Hvis ikke en '&' avslutter linjen skal foreldreprosessen vente til barneprosessen er ferdig.

Innebygde kommandoer

Som andre shell skal ifish ha enkelte innebygde kommandoer (i tillegg til *exit* og *quit*). Hvis man for eksempel i et vanlig shell prøver å finne hvilken versjon av *cd* man bruker ved å utføre kommandoen `type cd` får man beskjed at dette er en innebygget kommando i shellet:

```
[vizzini] 1 > type cd
cd is a shell builtin
```

Du skal derfor legge inn de innebygde kommandoene som er spesifisert under i shellet -- det vil si h (history) (`cd` skal ikke implementeres). **Disse skal ikke forkes ut eller eksekveres med execve()**.

`h` - eksekveringshistorie

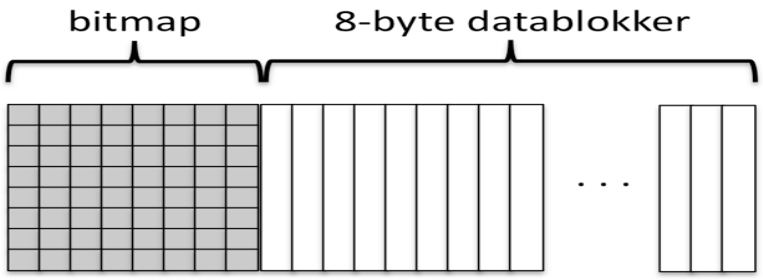
En vanlig innebygget kommando er *h*, som er et alias for *history*. Denne viser en oversikt over de siste kommandoene (inkludert parametere) shellet har utført. Du skal legge inn *h* som en innebygget kommando i **ifish** hvor shellet skal kunne holde de **n** siste kommandoene. Antallet **n** skal kunne variere avhengig av hvor mye minne som brukes per kommando (se under). Hvis brukeren så utfører *h* skal en liste over de siste kommandoene skrives ut hvor eldste kommando ligger øverst og den nyeste nederst (som vil være `h`) - a la:

```
paalh@ifish 10 > h

History list of the last 10 commands:
  10: cd
   9: ls
   8: cd oblig
   7: make
   6: man gcc
   5: man getenv
   4: h
   3: more ifish.c
   2: ifish
   1: h
```

Håndtering av minne for history

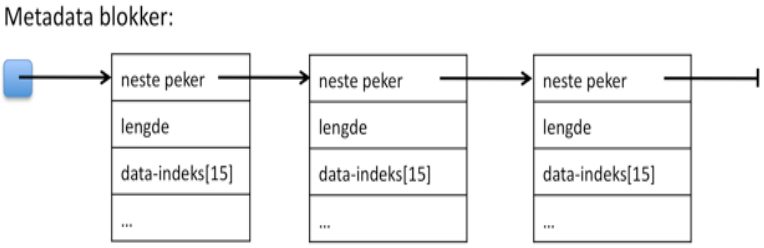
For å lagre kommandoene for *h*, skal du implementere et system som holdes i primærminnet. Dette skal være et kontinuerlig minne som skal ha plass til *64* **8-byte** blokker samt en bitmap som skal indikere hvilke blokker som er i bruk og ikke - det vil si at bit-verdiene skal være henholdsvis `0` hvis blokkene er ledig og `1` hvis blokker er opptatt. Et eksempel på hvordan minnet skal allokeres og deles opp er vist i figuren under.



Siden en kommando er antatt å være av vilkårlig lengde, men som nevnt over ikke overstige 120 tegn, kan en enkelt kommando trenge flere slike **8-byte** blokker (á la pages i minnet eller diskblokker på disken). Disse blokkene trenger ikke nødvendigvis ligge etterhverande. Når en ny kommando skrives (utføres av shellet), skal systemet lete etter ledige blokker i bitmapen, allokere de som trengs (sette bittene) og bruke de tilsvarende blokkene for å lagre kommandoen i disse minneblokkene. Hvis det ikke er nok ledige blokker for å lagre kommandoen i history, skal systemet kaste ut den eldste kommandoen fra listen, frigjøre dens allokerte datablokker, og igjen prøve å legge inn den nye (evt. gjenta for å skaffe tilstrekkelig nok plass). Når en kommando slettes fra listen, skal dens datablokker også slettes, dvs. fjerne innholdet i dem (HINT: Bruk for eksempel *bzero*).

Metadatastruktur

I tillegg trenger du en type metadatablokk for hver kommando som lagres i *history*. Metadataene skal legges inn i en struktur som i allefall må inneholde id-ene til datablokkene som brukes til å lagre kommandoen. Her kan man gjerne bruke "direktepekere" (her vil det si indekser til bitmapen og datablokk-minnet). Man trenger da 15 slike for å kunne lagre en kommando på opptil 120 tegn. I tillegg trenger man et "lengde" felt for å lagre informasjon om hvor lang kommandoen er. Videre skal metadatablokkene organiseres ved å bruke en lenket liste - den nyeste kommandoen skal ligge først i listen, og den eldste (og den som eventuelt skal fjernes for å frigjøre plass) skal ligge sist. Det vil si at datastrukturen også trenger en nestepeker til neste metadatablokk. (Hvis du trenger andre elementer i strukturen skal dette beskrives og argumenteres for.) Et eksempel på en slik liste er vist under.



`DEBUG 1:` Som debug-informasjon skal programmet skrive ut innholdet av bitmapen hvor du skriver ut 0-ere og 1-ere. Hver linje skal være på 32 tegn.

```
DEBUG - BITMAP:

10101010101010111100011001100111
11000110011001111010101010101011
```

`DEBUG 2:` Som debug-informasjon skal programmet skrive ut innholdet av datablokkene. Hver datablokk skal være avskilt med to "#" (dvs, "##"). Hver linje skal skrive ut *4* 8-byte blokker. (Merk at blokkene som er markert i bitmapen i eksempelet over IKKE stemmer med hvilke blokker som er brukt i eksempelet under).

```
DEBUG - DATABLOCKS:

##more ifi##sh.c    ##        ##cd oblig##
##gcc -DDE##BUG -o m##y_perfec##t_ifish ##
##        ##        ##        ##        ##
##ifish.c ##        ##        ##        ##
##        ##        ##        ##        ##
##h -d 5  ##        ##ls -al *##        ##
##.pdf    ##        ##        ##man gete##
##nv      ##        ##        ##        ##

. . . .

##h       ##h 5     ##make    ##        ##
```

Oppsummert vil dette bety at når en kommando skrives inn, skal shellet opprette en metadatablokk og legge denne inn i den listen. Lengden på kommandoen må regnes ut og ledige data blokker må allokeres, sette bittene i bitmapen og legge indeksene til datablokkene inn i "direktepekerene" i strukturen. Tilslutt må man ikke glemme og lagre kommandoen i datablokkene. Tilsvarende, når en kommando må slettes fra *history*, skal både datablokkene og metadatablokken frigjøres.

`h i` - utfør den i'te yngste historiekommandoen

Du skal utvide funksjonaliteten til `h` slik at hvis et tall *i* sendes som argument skal, ikke en liste skrives ut, men den *i*'te yngste kommandoen utføres. For eksempel, `h 3` vil i eksempelet over medføre at kommandoen `more ifish.c` blir utført.

`h -d i` - slette oppføring i kommandohistorien

Du skal utvide funksjonaliteten til `h` med opsjonen *-d* slik at hvis opsjonen -d oppgis med et tall *i*, skal den *i*'te yngste kommandoen slettes fra historien. For eksempel, `h -d 3` vil i eksempelet over medføre at kommandoen `more ifish.c` blir slettet. Pass på at resten av listen holdes inntakt selv om en kommando blir slettet.

Innlevering

Besvarelsen skal bestå av den *godt kommenterte* kildekoden til programmet og en makefile for kompilering. Under finner dere instruksjoner på hvordan dere skal levere inn, og **ved å levere inn oppgaven samtykker dere på at dere overholder reglene for innleveringen.**

Elektronisk innlevering

Oppgaven skal leveres elektronisk, dvs. at ingen papirkopi er nødvendig. Dere skal opprette en katalog med deres *brukernavn som navn*. Denne katalogen skal inneholde alle filer som skal leveres (kode og makefile). Denne katalogen lager dere en tar-ball (`tar`) av og komprimerer med `gzip`. Tar-ballen skal navngis med brukernavnet og med postfix `.tgz` - for eksempel *paalh.tgz* - og lastes opp på Devilry **før** tidsfristen utløper.

Regler for innlevering og bruk av kode

Ved alle pålagte innleveringer av oppgaver ved Ifi enten det dreier seg om obligatoriske oppgaver, hjemmeeksamen eller annet forventes det at arbeidet er et resultat av studentens egen innsats. Å utgi andres arbeid for sitt eget er uetisk og kan medføre sterke reaksjoner fra Ifis side. Derfor gjelder følgende:

1. Deling (både i elektronisk- og papirform) eller kopiering av hele eller deler av løsningen utviklet i forbindelse med de obligatoriske og karaktergivende oppgavene i kurset er ikke tillatt.

2. Deling/distribuering av kode og oppgavetekst med personer som ikke er eksamensmeldt i inf1060 dette semesteret, med unntak av kursledelsen og gruppelærere, er ikke tillatt.

3. Hente kode fra annet hold, f.eks. fra andre ``open source" prosjekter eller kode funnet på nettet er ikke tillatt.

4. Det er greit å få generelle hint om hvorledes en oppgave kan løses, men dette skal eventuelt brukes som grunnlag for egen løsning og ikke kopieres uendret inn.

5. Kursledelsen kan innkalle studenter til samtale om deres innlevering.

Reglene om kopiering betyr ikke at Ifi fraråder samarbeid. Tvert imot, Ifi oppfordrer studentene til å utveksle faglige erfaringer om det meste. Vi oppfordrer til at studentene skal kunne lære av hverandre, men, som sagt, man skal ikke dele/distribuere/kopiere noen form for kode. Det som kreves er som nevnt at man kan stå inne for det som leveres. Hvis du er i tvil om hva som er lovlig samarbeid, kan du kontakte gruppelærer eller faglærer.

Se ellers IFIs retningslinjer

```
Lykke til!
    Michael, Tor og Pål
```

Vedlegg 1: safefork.c

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/errno.h>

extern int errno;

#define MAX_PROCESSES 6

static int n_processes(void)
{
  return system("exit `/bin/ps | /store/bin/wc -l`")/256;
}


pid_t safefork(void)
{
  static int n_initial = -1;

  if (n_initial == -1)  /* Første gang funksjonen kalles: */
    n_initial = n_processes();
```

```
  else if (n_processes() >= n_initial+MAX_PROCESSES) {
    sleep(2);
    errno = EAGAIN;  return (pid_t)-1;
  }

  return fork();
}
```

NOTE: Stien til programmene `ps` og `wc` kan være forskjellig. Sjekk dette med kommandoen `which ps` eller `which wc`.