HØGSKOLEN I BERGEN
Avdeling for ingeniørutdanning
Data

Øving 9 (Obligatorisk)

Kommentarer til øvingen:

- Utlevert: 29. oktober 2010.
- Øvingen gjøres på lab A516 eller A521:
  - Halve klassen møter torsdag 4. november kl. 14:05 til 15:45.
  - Resten av klassen møter fredag 5. november kl. 12:15 til 13:55.
- Innleveringsfrist er onsdag 10. november kl. 14:00.
- Øvingen leveres på It's Learning.
- Øvingen utføres i grupper på 3-4 studenter.
- Oppgave 1 og 5: Godkjennes på lab'en hvis tid (avkrysning), ellers innlevering tilsvarende oppgave 2, 3 og 4.
- Oppgave 2, 3 og 4: Innlevering på papir og på It's learning.
- Innlevering vil ikke bli godkjent hvis den inneholder filer foreleser har problem med å lese:
  - Unngå norske (eller andre ikke-amerikanske) tegn i filnavn.
  - Spesifiser benyttet tegnsett for txt-dokumenter.

Formål: Bli kjent med en del fundamentale begreper

Oppgave 1

Ved å bruke systemkall mot en Linux-maskin skal dere lage et C++-program som gjør følgende.

Programmet skal be om og lese et filnavn fra tastaturet.

- Dersom filen eksisterer vises innholdet på skjermen.
- Dersom filen ikke eksisterer startes en barneprosess. Barneprosessen skal starte en teksteditor som lar bruker skrive inn tekst i filen. Når barneprosessen avsluttes skal foreldreprosessen vise innholdet i filen

For å hjelpe dere å løse oppgaven har jeg laget en mengde eksempler på C++ program som gjør C-systemkall på en Linux-maskin. Eksemplene finner dere på It's-Learning.

Programeksempler

finnesA.cpp

Dette eksempelet viser hvordan vi enkelt kan undersøke om en fil finnes.

finnesB.cpp

Dette er en alternativ metode for å sjekke om filen finnes. Denne gjør akkurat det samme som *finnesA.cpp*, men metoden bruker systemkall og er dermed ikke portabel til andre OS.

Denne metoden kan være et utgangspunkt for en mer forseggjort kontroll der vi også undersøker skrivetilgang til mappen der vi ønsker å lagre filen.

Aktuelle manualsider:

- `stat(2)`

ov1a.cpp

I dette eksempelet benyttes systemkallet `execl()` til å gi kjørende prosess et nytt innhold.

Aktuelle manualsider:

- `execl(3)`
- `execve(2)`

ov1b.cpp

Systemkallet `vfork()` oppretter en barneprosess. Det brukes minimalt med ressurser for å opprette barneprosessen. Systemkallet er ment å brukes når barneprosessen øyeblikkelig skal erstattes med nytt innhold.

I dette eksempelet kjører foreldreprosessen og barneprosessen parallelt. Barneprosessen erstattes med Emacs.

Aktuelle manualsider:

- `execl(3)`
- `execve(2)`
- `vfork(2)`
- `sleep(3)`

ov1c.cpp

I dette programmet venter forelderen på barneprosessen ved å bruke systemkallet `wait()`. Når barneprosessen avsluttes overføres barnets exit-status til forelderen som først da fortsetter.

Barneprosessen opprettes ved systemkallet `fork()`. Barnet får da en kopi av forelderens hukommelse. For variabler som er gitt verdi før kallet av `fork()` vil begge prosessene se verdien. For endringer i data som gjøres etter `fork()` vil endringen kun gjelde for den prosessen som gjør endringen.

Aktuelle manualsider:

- `fork(2)`
- `sleep(3)`
- `_exit(2)`
- `wait(2)`

ov1d.cpp

For å kommunisere data mellom foreldreprosess og barneprosesser kan vi bruke delt hukommelse, eller vi kan sende beskjeder mellom prosessene.

Delt hukommelse vil vi ta i en senere oppgave. For å sende beskjeder mellom prosessene har vi mange metoder. I denne oppgaven benytter vi en *pipe*, *rør* eller *kanal*. Røret er rettet, dvs at vi sender inn data i den ene rørenden og data kommer så ut i den andre rørenden.

Da pipe'en er opprettet før kallet av `fork()` ser begge prosessene pipe'en og vi kan bruke den til å sende data mellom prosessene.

I dette eksempelet venter forelderen på barnet ved hjelp av `wait()`. Barnet leser en tekst som sendes inn i pipen. Når barnet avsluttes og forelderen så starter leser forelderen teksten fra andre enden av pip'en.

Aktuelle manualsider:

- `pipe(2)`
- `fork(2)`
- `write(2)`
- `wait(2)`
- `read(2)`

ov1e.cpp

Dette eksempelet er en variasjon over «ov1d.cpp». I dette eksempelet erstattes barneprosessen med nytt innhold. Det nye innholdet i barneprosessen skriver tekst til STDOUT. Dette fanges opp og sendes inn i pipe'en som foreldreprosessen leser fra når den starter opp igjen etter at barnet er avsluttet.

STDOUT er knyttet til fil-deskriptor 1. Ved systemkallet dup2() kan vi koble en gammel fildeskriptor mot en ny. Kallet dup2(pipefd[1],1) vil slette deskriptor 1 som STDOUT er koblet mot. I stedet opprettes det en ny deskriptor 1 som er identisk med deskriptoren pipefd[1]. Når nå utskrift sendes til STDOUT vil den gå inn i rørenden pipefd[1].

Foreldreprosessen lager tilsvarende en kopi, fildeskriptor 0, av pipefd[0]. Når forelderen leser fra STDIN leses fra fildeskriptor 0 som nå er identisk med pipefd[0].

Aktuelle manualsider:

- `pipe(2)`
- `fork(2)`
- `write(2)`
- `wait(2)`
- `read(2)`
- `dup2(2)`

Oppgave 2

Hva er de viktigste oppgavene for et operativsystem?

I generelle større operativsystem kan vi ofte få utført både sanntidskjøring og interaktiv kjøring. Gi en kort forklaring på hver av disse typene.

Oppgave 3

For å utnytte maskinressursene godt er operativsystemet som oftest laget for å kunne ha mange prosesser i gang samtidig (multiprogrammering). Det oppstår da behov for beskyttelse (*protection*) av de enkelte ressurser. Forklar hva som menes med beskyttelse av

- inn/ut - operasjoner (*I/O – operation*),
- indre lager,
- CPU.

Forklar, i prinsippet, hvordan hver av de tre nevnte ressursene blir beskyttet.

Oppgave 4

Gjør følgende oppgaver fra læreboken (Operating System Concepts):

- ”Describe the differences among short term, medium turn and long term scheduling”. (Oppgave 3.1 fra lærebok utg. 7.)
- Oppgave 4.3 fra lærebok utg. 8 (tilsvarer 4.4 fra lærebok utg 7.)

Oppgave 5

Dere skal lage et C++-program som benytter tråder implementer ved POSIX Pthreads.

Programmet skal inneholde en datastruktur svarende til den 2-dimensjonale tabellen under:

$$A = \begin{bmatrix} 12 & 13 & 17 & 21 \\ 21 & -9 & 18 & 33 \\ -12 & 22 & 2 & 0 \\ 2 & 55 & 9 & -5 \end{bmatrix}$$

Programmet skal summere alle tallene i tabellen. Summeringen skal utføres ved hjelp av to tråder som hver for seg finner totalsummen.

Den ene tråden skal finne summen av alle tallene ved å summere siffrene rekke for rekke. Den andre tråden skal finne totalsummen ved å summere siffrene kolonne for kolonne.

Den tråden som først finner svaret skal kansellere den andre tråden som ikke lengre trengs. Foreldretråden skal så presentere svaret på skjermen.

Summering rekkevis

Den ene barnetråden skal summere sifrene rekkevis. Dvs:

Sum rekke1 + sum rekke 2 + sum rekke 3 + sum rekke 3 + sum rekke4
= (12+13+17+21) + (21□9+18+33) + ⇦

Summering kolonnevis

Den andre barnetråden skal summere sifrene kolonnevis. Dvs:

Sum kolonne 1 + sum kolonne 2 + sum kolonne 3 + sum kolonne 4
= (12+21□12+2) + (13□9+22+55) + ⇦

Programeksempler

For å hjelpe dere å løse oppgaven er det laget to C++ program. De benytter POSIX Pthreads og er laget for Linux. Programmene finner dere på *It's Learning*.

Programmene må lenkes mot biblioteket *pthread*.

ov1f.cpp

Eksempelet viser hvordan tråder opprettes, og hvordan forelderen settes til å vente til barnetråden avsluttes.

Kompiler programeksempelet med

```
g++ ov1f.cpp -o ov1f -lpthread
```

I dette eksempelet opprettes en ny tråd med `pthread_create()`. Foreldretråden venter til barnet er ferdig ved bruk av `pthread_join()`. Barnetråden endrer på felles data og returnerer sin exit-status til forelderen.

Aktuelle manualsider:

- `man pthread.h(0p)`
- `man errno(3)`
- `man pthread_create(3p)`
- `man pthread_join(3p)`

- `man perror(3)`
- `man pthread_exit(3p)`

ov1g.cpp

Eksempelet viser bruk av både asynkron og utsatt kansellering.

Kompiler programeksempelet med

```
g++ ov1g.cpp -o ov1g -lpthread
```

I dette eksempelet opprettes først en tråd. Den kanselleres så av foreldretråden ved asynkron kansellering. Barnet setter sin kanselleringsmetode ved `pthread setcancelstate()`. Standard oppførsel ved kansellering er utsatt kansellering, så asynkron kansellering må angis eksplisitt. Forelderen kansellerer barnet ved bruk av metoden `pthread_cancel()`.

Etter dette opprettes enda en ny tråd. Denne kanselleres av forelderen ved utsatt kansellering. Etter at forelderen har benyttet `pthreead_cancel()` venter forelderen til barnet lar seg kansellere ved `pthread_join()`. Barnet angir et kanselleringspunkt ved å bruke `pthread_testcancel()`.

I tråden som skal kanselleres ved utsatt kansellering bør du unngå å benytte `cout`, `cin` eller `cerr` når `pthread_cancel()` benyttes mot tråden. Ellers kan programmet låse seg eller krasje.

En tråd kan kanselleres av forelderen, men en tråd kan også kansellere sine søsken.

Kanselleringspunkt kan settes ved `pthread_testcancel()`, men også mange andre systemkall gir kanselleringspunkt, deriblant `sleep()`. Dette er grunnen til at eksempelet benytter en egen `vent()` metode. Du må kanskje korrigere argumentet til `vent()` for å få fornuftige ventetider.

Dersom du ønsker å benytte `sleep()` eller andre metoder som gir kanselleringspunkt kan du eksplisitt slå av kansellering ved å benytte `pthread_setcancelstate()`.

Aktuelle manualsider:

- `man errno(3)`
- `man pthread_create(3p)`
- `man pthread_join(3p)`
- `man perror(3)`
- `man pthread_exit(3p)`
- `man pthread_cancel(3p)`
- `man pthread_setcancelstate(3p)`
- `man pthread_setcanceltype(3p)`
- `man pthread_testcancel(3p)`