Övning 1: Parallella processer i *C* med `fork()` och systemprogrammering

Dessa övningar är tänkta att utgöra förberedelseövningar inför laboration 1. Denna övning behandlar parallella processer och systemprogrammering. Dessa övningar är tänkta att vara ganska lätta att komma igenom, det som ska vara mer utmanade är själva laborationen. Er strävan bör vara att snabbt klara av övningarna för att kunna ägna mycket energi åt själva laborationen. Börja mycket gärna med laborationen så fort ni förstår lite av vad som ska göras även om ni inte är klara med övningarna. Det är bara laborationen som redovisas.

Innehåll:
* Kommandoradsargument
* Skapa barnprocesser
* Hantera zombier
* `execl()`

Relevanta manualsidor: `fork(2)`, `getpid(2)`, `wait`, `sleep`, `exit`, `execl`, `exec`, `execve`.

Uppgifter

1. Kommandoradsargument

När man kör ett kommando i skalet kan man skriva argument efter kommandot. Till exempel kan man skriva `ls -l` och då är det `ls` (fillistning) som körs och argumentet här är ett "`l`". (Med "`-`" framför.) Vi ska lära oss göra detta i *C*. Det finns en utmärkt genomgång av detta i *Advanced Linux Programming*. Vi ger ett exempel som illustrerar:

Program:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  printf("Argument:\n\n");
  for(i=0;i<argc;i++)
    printf("%d: %s\n", i, argv[i]);
  printf("\n\n");
}
```

Körning:

```
> ./kommradarg 1 2 3 olle lisa
Argument:

0: ./kommradarg
1: 1
2: 2
3: 3
4: olle
5: lisa
```

Programmet heter, efter kompilering `kommradarg` och tar hur många argument som helst. Skriv in det och testa. Modifiera programmet så att det heter "summera" och så att det summerar de ingående argumenten som vi antar är heltal. (Ledning: Att omvandla en sträng till heltal görs bäst med funktionen `atoi()` som finns i `<stdlib.h>` som alltså också måste inkluderas.)

2. Skapa barnprocesser

a) Skriv om programmen `fork1.c`, `fork2.c`, `fork3.c`, `fork4.c`, `fork5.c`, `fork6.c`, `fork7.c` och `zombiecreator.c` från föreläsningen och inkludera ordentliga kontroller av systemanrop. Testa att de fungerar som på föreläsningen. (Anmärkning: du behöver givetvis **inte** skriva in alla om du inte känner att du behöver det.)

b) Skriv nu ett program som tar ett heltal som argument (på kommandoraden) och som skapar så många barnprocesser. Alla barnprocesser ska köra samtidigt och skriva ut sina PID och föräldrarnas PID och sedan och göra något trivialt, kanske bara `sleep(20)`. Alla barnprocesser ska inväntas. Starta programmet i bakgrunden och använd kommandot `ps -elfa | grep <ditt användarnamn>` för att få upp en lista över alla körande processer. Alla systemanrop ska som vanligt kontrolleras.

c) Modifiera programmet i b) så att det tar två argument, ett är antal barnprocesser som ska skapas, och det andra argumentet ska ange vilken av barnprocesserna som ska skapa ytterligare en barnprocess, (ett barnbarn alltså.) Se till att du kan kontrollera att programmet fungerar genom att inkludera anrop till `sleep()` som möjliggör att du kan se processernas ingående släktförhållanden (vem är förälder till vem) med `ps -elfa | grep <ditt användarnamn>`. Ett anrop till programmet kan se ut så här: `>./createchildrenandonegrandchild 10 5 &`. Detta anrop har två kommandoradsargument, 10:an anger att det är 10 barn som ska skapas och 5:an anger att det femte barnet ska skapa ytterligare ett barn. (`&`-tecknet är för att köra programmet i bakgrunden.) Programmet ska kontrollera att argumentet som anger vilket av barnen som ska skapa ytterligare ett barn verkligen anger ett av de körande barnen. Att anropa programmet med argumenten "10 12" är meningslöst eftersom det inte finns 12 barn. Programmet ska då inte köra utan ge ett felmeddelande. Likaså ska felmeddelande anges om fel antal argument anges.

3. Hantera zombier

Utvidga programmet från 2:an till att ta ett tredje argument som anger hur många av barnen som ska bli zombier direkt. Det barn som ska skapa ytterligare ett barn får förstås inte bli zombie direkt, det ska ju kunna invänta sitt barn. Kontrollera att programmet fungerar ordentligt med "`ps -elfa | grep ...`" som ovan.

4. `execl()`

a) I `fork7.c` finns ett anrop till `execl()`. Det är ett systemanrop och returvärdet måste alltså kontrolleras, systemanrop kan ju gå fel. Skriv om det programmet och ta hänsyn till att det kan gå fel vid anropet till `execl()`. Modifiera sedan även anropet så att systemanropet verkligen går fel och se till att ditt program kan hantera detta. (Ett lätt sätt kan vara att skriva fel i filnamnet i parametern till `execl()` som anger vilken barnprocess som ska startas.)

b) Läs igenom manualsidorna till `execl()` och `execve()` och försök förstå så mycket du kan där. Förstå särkilt skillnaden mellan att göra ett anrop till `fork()`, som skapar en barnprocess, och att anropa `execl()`, som inte skapar en barnprocess, utan ersätter den anropande processen med den som man begär ska köras via `exec()`. Kolla gärna upp `vfork()` också.
c) Skriv ett program som med hjälp av `execlp()` startar ett annat *C*-program (som du också skriver)

Teoriavsnitt (läs igenom före laborationen, skriv in och testa olika program särskilt programmen med `fork()`.): *Advanced Linux Programming* kap 1.2, 1.5, 2.1, 2.2, 3.1, 3.2, 3.3, 3.4.