Övning: Början till IPC via *Pipes* samt en del om `execlp()`

Denna övning är speciellt uformad som förberedelse inför kursmöte 10. Man behöver inte fullborda övningen, men det är viktigt att ha *börjat* med den och gjort ett bra försök, om man hinner klart är det förstås ännu bättre. Se dock säkert till att börja på de båda delarna av denna övning, det om fildeskriptorer och pipe men också det om `execlp()`.

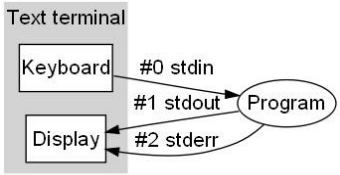
Från wikipedia.org: (http://en.wikipedia.org/wiki/File_descriptor):
"In computer programming, a **file descriptor** is an abstract key for accessing a file. The term is generally used in POSIX operating systems. In Microsoft Windows terminology and in the context of the C standard I/O library, "file handle" is preferred, though the latter case is technically a different object (see below).
In POSIX, a file descriptor is an integer, specifically of the C type `int`. There are 3 standard POSIX file descriptors which presumably every process (save perhaps a daemon) should expect to have:

Integer value	Name
0	Standard Input (stdin)
1	Standard Output (stdout)
2	Standard Error (stderr)

Generally, a file descriptor is an index for an entry in a kernel-resident data structure containing the details of all open files. In POSIX this data structure is called a file descriptor table, and each process has its own file descriptor table. The user application passes the abstract key to the kernel through a system call, and the kernel will access the file on behalf of the application, based on the key. The application itself cannot read or write the file descriptor table directly.
In Unix-like systems, file descriptors can refer to files, directories, block or character devices (also called "special files"), sockets, FIFOs (also called named pipes), or unnamed pipes.
The `FILE *` file handle in the C standard I/O library routines is technically a pointer to a data structure managed by those library routines; one of those structures usually includes an actual low level file descriptor for the object in question on Unix-like systems. Since *file handle* refers to this additional layer, it is not interchangeable with *file descriptor*."

Mer om fildeskriptorer

Operativsystemet (UNIX/POSIX) har en tabell med fildeskriptorer för alla öppna filer. I fildeskriptortabellen hanteras också körande processers kommunikation via STDIN och STDOUT då data flödar till och från det som STDIN och STDOUT pekar på. För ett program som man kör från en kommandoprompt ser det ut så här:



(Vi har tidigare refererat till STDIN och STDOUT som *standard in* och *standard out*.) När man skriver vid en kommandoprompt och använder `|`-tecknet (kallas pipe=rörledning) så kopplas den ena processens `STDOUT` på den andras `STDIN` och man kan på så sätt ta resultatet från en process och skicka in i en annan, så här kan det se ut:

```
$ ls -l | grep Johnny
```

Här är process 1 = kommandot `ls -l` och process 2 är kommandot `grep Johnny`. Resultatet är att utmatningen från "`ls -l`" (en fillistning) skickas in i processen "`grep Johnny`" som sorterar fram de rader som innehåller texten "`Johnny`". Det sammansatta resultatet är två processer som tillsammans sorterar fram de fillistningsrader som innehåller texten "`Johnny`". Det kan vara ett sätt

att ta fram alla filer som tillhör användaren `Johnny` eller har `Johnny` i sitt filnamn. Här ska vi börja med att lära känna hur vi kan åstadkomma detta i *C*. Vi har också, i och med kursmöte 3, studerat hur vi kan få ett *C*-program att skriva till strömmen *standard out* (genom `write(1,&a,sizeof(int))` och läsa från strömmen *standard in* (genom `read(0,&a,sizeof(int))`. Det var de här experimenten på kursmöte 3 där vi arbetade med kommandon av typ `./out 3 | ./spawn | ./showp` där `out`, `spawn` och `showp` var *C*-program som styrdes genom att vi läste/skrev från/till STDIN/STDOUT.

Relevanta kommandon som ni kan undersöka (med `man`): `fork()`, `pipe(2)`, `pipe(7)`, `close(2)`, `open(2)`, `read(2)`, `write(2)`, `getpid()`, `isalpha()`, `mkfifo`, `unlink`. (`read(2)` betyder att man ska skriva `man 2 read`, alltså `2`:an ska in mellan `man` och `read`, då hittar man sektion 2 i manualsidorna som beskriver hur `read` fungerar i *C*. Om man inte skriver en 2:a så får man `read` som hör till *bash*.)

Teoriavsnitt: skriv in och testa olika program särskilt programmen med `fork()` och `pipe()`. Advanced Linux Programming kap 3.1, 3.2, 3.4, 5.4.

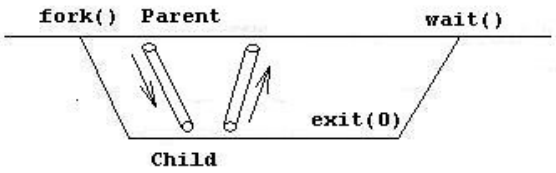
Rörledningar/Pipes i *C*

En föräldraprocess som skapat en barnprocess kan kommunicera med barnprocessen via en så kallad *pipe*. (På svenska skulle man kunna säga "rörledning", men jag väljer att säga pipe som uttalas "pajp".) Vi skapar en pipe i *C* med systemanropet `pipe()` och det kan se ut så här:

```
int fds[2];
pipe(fds);
```

Det övre är en deklaration (en array av heltal) och det undre är systemanropet `pipe()` som körs på denna array, innan en pipe kan finnas måste vi alltså ha en liten heltalsarray med två platser, ovan `fds[0]` och `fds[1]`, och när man gör `pipe()` på den talar man om för operativsystemet att man tänker använda arrayen som åtkomstpunkt till en pipe som då skapas av operativsystemet. Då skapar systemet pipen och innehållet i arrayen blir två *fildeskriptorer*, den ena kan man skriva till `fds[1]` och den andra kan man läsa från `fds[0]`. Ofta kommer vi att kalla `fds[1]` för "skrivänden" och `fds[0]` för "läsänden". (0=läs och 1=skriv, så är det *alltid* med pipes.)

Det är teoretiskt möjligt för en process att skriva till `fds[1]` (skrivänden) och läsa från `fds[0]` (läsänden) för att läsa det som den nyss skrivit, men det är inte så meningsfullt. Det är vanligare att låta en pipe bli en kommunikationskanal mellan två olika processer där den ena processen skriver i skrivänden på pipen och den andra läser från läsänden. På så sätt kan man upprätta kommunikation i ena riktningen. Om man vill ha kommunikation i båda riktningarna **måste** man ha två pipes. Vi kan illustrera det i en tidsdiagram så här:



Här ser vi en föräldraprocess som skapat ett barn. Mellan dem finns två olika pipes för kommunikation i båda riktningarna (en för varje håll). Vi ska nu se hur det fungerar programmeringsmässigt. Det första som krävs är förstås arrayen med fildeskriptorer (ovan kallad

`fds`). Efter man deklarerat den arrayen och gjort `pipe()` på den kan man göra `fork()`. Som vi minns kommer då både förälder och barn att ha *varsinn* uppsättning av de fildeskriptorer som indikerar pipen. Då en process gör `fork()` så kopieras ju innehållet i alla variabler till barnet som har en egen uppsättning av alla förälderns variabler. Då kommer alltså både föräldern och barnet ha *varsinn* läsände och *varsinn* skrivände i deskriptorerna `fds[0]` och `fds[1]`. Men de kommer fortfarande att leda till samma pipe, själva pipen ligger i en struktur som operativsystemet håller reda på. Innan kommunikation kan ske måste därför ordning skapas här: två processer bör inte ha en deskriptor öppen, det går men det skapar oreda i den struktur vi vill utveckla här. Därför måste den process (av barnet och föräldern) som ämnar skriva stänga sin läsände och den process som ämnar läsa måste stänga sin skrivände. Koden kommer då att principiellt se ut så här:

pipe()
fork()

barnet:
stänger skrivänden
läser sedan från läsänden tills allt är klart.

föräldern:
stänger läsänden
skriver sedan till skrivänden tills allt är klart.

När det som ska göras är klart så stänger även föräldern skrivänden och när barnet på nytt försöker läsa därifrån så misslyckas den läsningen och på det sättet kan barnet förstå att kommunikationen är avbruten eller fullbordad. Man läser och skriver med systemkommandona `read()` respektive `write()` och de returnerar antal tecken som lästs eller skrivts. Alltså efter raden som det står "*läser sedan från läsänden tills allt är klart*" (i barnet) så kan alltså barnet stänga sin läsände och avsluta sig. Vidare, efter raden som det står "*skriver sedan till skrivänden tills allt är klart*" (i föräldern) så kan alltså föräldern stänga sin skrivände och invänta barnet. Vi ska se på hur detta kan se ut i ett programexempel. (här finns inte den eleganta avslutningen med, det ingår i denna övning att du ska ordna en elegant avslutning.)

En funktion som heter `countPrimes()` anropas av en barnprocess, föräldern och barnet ska kommunicera via två pipar (”*pajpar”*) och barnet använder `countPrimes()` för att utföra en beräkning som då avlastar föräldern.

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#define READ  0
#define WRITE 1

/* Countprimes: Count the number of primes up to upTo */
int countPrimes (int upTo) {
  int number, count, factor, factorFound;
  for(number=2,   count=0;number<=upTo;number++)  {
    for(factor=2,factorFound=0; factor*factor<=number;factor++)
      factorFound = (number%factor==0);
    if(!factorFound)count++;
  }
  return count;
}
```

```
int main()
{
  int request[2], response[2], value, result;

  if(pipe(request)<0 || pipe(response)<0) {
    perror("Can't create pipes\n");
    return 1;
  }

  switch(fork())
  {
    case -1: /* FAIL */
      perror("Cannot fork()");
      return 1;

    case 0: /* CHILD PROCESS - CONSUMER */
      close(READ);
      close(WRITE);
      close(request[WRITE]);
      close(response[READ]);
      for(;;) {
        read(request[READ], &value, sizeof(value));
        result = countPrimes(value);
        write(response[WRITE], &result, sizeof(result));
      }
      break;

    default: /* PARENT PROCESS - PRODUCER */
      close(request[READ]);
      close(response[WRITE]);
      for(;;) {
        printf("Ange ett tal: "); scanf("%d", &value);
        if(sizeof(value)!=write(request[WRITE],&value,sizeof(value)))
        {
          perror("Cannot write thru pipe.\n");
          return 1;
        }
        read(response[READ],&result,sizeof(result));
        printf("Result: %d.\n",result);
      }
      break;
  }
}
```

Föräldern accepterar inmatade heltal skickar detta inmatade heltal till ett barn som räknar ut hur många primtal det finns upp till och med det inmatade talet. Resultatet skickas sedan tillbaka till föräldern i en annan pipe.

Övningsuppgift: Skriv in detta program och testa så att det fungerar. Ordna ett snyggt avslut genom att man får mata in "0" för att avsluta. Då ska föräldern montera ner körningen på det sätt som beskrivits ovan. Var noga med att både barn och förälder stänger ALLA fildeskriptorer när de inte behövs. Detta inkluderar att de ska stängas när barnet och föräldern kört klart. Rita även ett tidsdiagram som illustrerar förhållandet mellan föräldra- och barnprocessen under hela deras körningar. Lägg också märke till systemprogrammeringsstilen ovan, vi kollar alltid returvärden från systemanropen `pipe()` och `fork()`. Diskutera gärna denna stil med en kamrat. När vi kommer djupare in i systemprogrammeringen ska vi även kolla resultatet från `close()` mm.

Provkörning av programmet som det fungerar nu:

```
Ange ett tal: 40
Result: 26.
Ange ett tal: 10
Result: 5.
Ange ett tal: 0
Result: 0.
Ange ett tal: ^C
$
```

Det som är fetmarkerat, alltså **`40`**, **`10`**, **`0`** och **`^C`** är det som anänvdaren gör, användaren måste alltså trycka *control-C* för att avbryta. Vad vi vill är att skriva om programmet (det behövs inte stora omskrivningar) så att både barnet och föräldern avslutar på ett snyggt sätt om användaren matar in talet **`0`** som ovan.

Att fundera kring: Variablerna `request[2]` och `response[2]`, finns i huvudprogrammet. Hur många minnespositioner tar de upp totalt då båda processerna kör? Motivera ditt svar väl. Vi kommer att behandla liknande problemställningar på kursmötet, det är ett centralt tema när det gäller IPC (*Inter-Process Communication*).

Rekapitulering av systemanropet `execlp()`

Vi har i tidigare föreläsningar nämnt **`execlp()`**. Vi fyller dock i dessa detaljer igen och formulerar dem på ett lite annorlunda sätt för ökad förståelse.

Vi har ovan sett hur kommunikation går till då man formerar processer med maskinkod innehållna i samma exekverbara fil. Vi ska nu se hur man kan starta processer med maskinkod från en annan fil. Den övergripande gången är att man skapar en barnprocess med `fork()` precis som vanligt, men sedan gör man systemanropet `execlp()` (eller något annat anrop i `exec()`-familjen, alla är egentligen varianter av det som kallas `execve()`. Jag brukar dock bara kalla dem `exec()`-anrop eller liknande). Vid varje användning av något av systemanropen i `exec()`-familjen ska parametrar levereras. Dessa parametrar är olika för olika anrop i `exec()`-familjen, för `execlp()` ser det ut så här:

```
execlp(”/bin/ls”,”ls”,”-l”NULL);
```

om vi vill vill köra kommandot `ls -l`. Det som händer då är att den process som anropar execlp() blir helt utbytt mot den process som anges i parametrarna. Om vi skriver ett program som ser ut så här:

```
#include <unistd.h>

main()
{
  execlp("/bin/ls", "ls", "-l", NULL);
}
```

och kör det så sker exakt samma sak som om vi ger kommandot `ls -l`. Det vi nu är intresserade

av är att köra `exec()`-anropet i en barnprocess så att barnet byts ut men föräldern är kvar. Vi kommer då att studera program med formen

```
pid_t pid; pid = fork();
if(pid==0)
{ execlp(); } // Bara barnet utför denna del
else
{ föräldrakod } // Bara föräldern utför denna del
```

Om anropet till `execlp()` lyckas så ersätts barnprocessen som skapats med `fork()` helt och hållet av den nya process som man anger genom att i parametrarna till `execlp()` ange ett program. Det enda som barnprocessen har gemensamt med den nya process som startar är processidentifikationsnumret `PID`. Detta betyder att `execlp()`-anropet inte ger något returvärde. För att kontrollera resultatet av den process som `execlp()` startar måste man använda `wait4()`. Om anropet till `execlp()` däremot misslyckas så returnerar `execlp()` värdet -1. Då kan man skriva felhanteringskod men den koden få då komma precis efter anropet till `execlp()`. (Vid det andra semikolonet ovan.)

Tidigare hade vi formen

```
int pid;
pid = fork();
if(pid==0)
  { barnprocess (det som bara barnet ska utföra) }
else
  { föräldraprocess (det som bara föräldern ska utföra) }
```

och då var vi tvungna att skriva den kod som barnet skulle utföra där det står ”`barnprocess`”. Med anrop ur `exec()`-familjen kan vi alltså starta barnprocesser med kod som hämtas externt. Någon annan kan alltså ha skrivit det program som vi då kör i en barnprocess. Det är mycket användbart i samband med processkommunikation. Provkör exemplet

```
#include <unistd.h>

main()
{
  execlp("/bin/ls", "ls", "-l", NULL);
}
```

ovan och skriv om det så att vi har en barnprocess som skapas och kös `ls -l` i. Kan du utvidga detta program så att du har en meny och väljer att antingen köra ls -l eller ls? Eller kanske något tredje kommando, ”`mkdir kalle`” till exempel?

Förberedelse 1: Studera exemplen med `exec()`-anropen i *Advanced Linux Programming*.
Förberedelse 2: När ett *Linux/UNIX/POSIX*-system startar skapas en process som har `PID` = 1 och den lever så länge systemet kör. Från den här processen skapas alla andra processer med hjälp av systemanropet `fork()`. Vad heter denna process? TIPS: kommandot `ps(1)` med lämplig parameter.