Möte 10: Kommunikation mellan processer - (*PI*)

Målet med att köra flera processer är ofta att få dem att samverka, som sagt, men för att få dem att samverka måste de ofta kommunicera. Vi ska studera processkommunikation med olika hjälpmedel, främst så kallade *pipes* och så kallade *sockets*.

Rörledningssystem, pipes

En pipe är enkelriktad kanal för kommunikation mellan 2 eller flera processer, det är den vi studerat i den förberedande övningen för detta kursmöte. En pipe kan ses som en FIFO-buffert (alltså en kö) och två fildeskriptorer som anger ingång och utgång för skrivning respektive läsning. En eller flera processer kan läsa från eller skriva till en pipe via systemanropen `read()` och `write()` med motsvarande fildeskriptorer för ut- respektive ingång till och från pipen. Vi ska nu se lite mer under huven vad som finns.

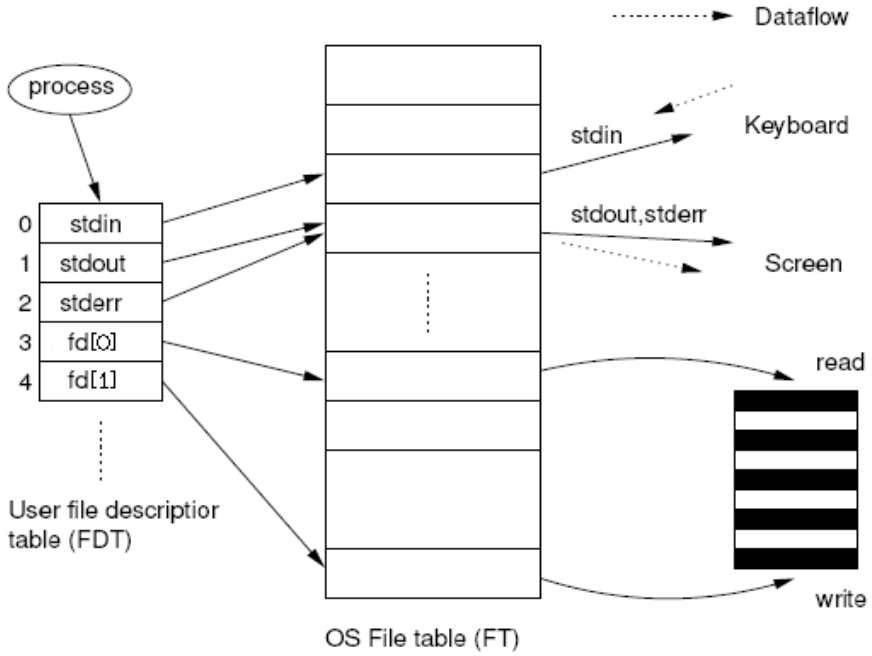
Systemanropet `pipe()`

För att allokera en pipe (alltså en rörledning) används systemanropet `pipe()`. Som vi sett i övningen anropas `pipe()` på en integerarray med två platser. Arrayen kommer att innehålla två fildeskriptorer, en för läsning och en för skrivning.

Exempel:

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

Efter ett lyckat anrop av detta slag är `fd[0]` *läsdeskriptor* och `fd[1]` är *skrivdeskriptor*. Som returvärde anges 0, för att ange ett lyckat anrop och – 1 anges för misslyckande. I figuren nedan visas datastrukturerna som uppkommer efter lyckat anrop av `pipe(fd)`.



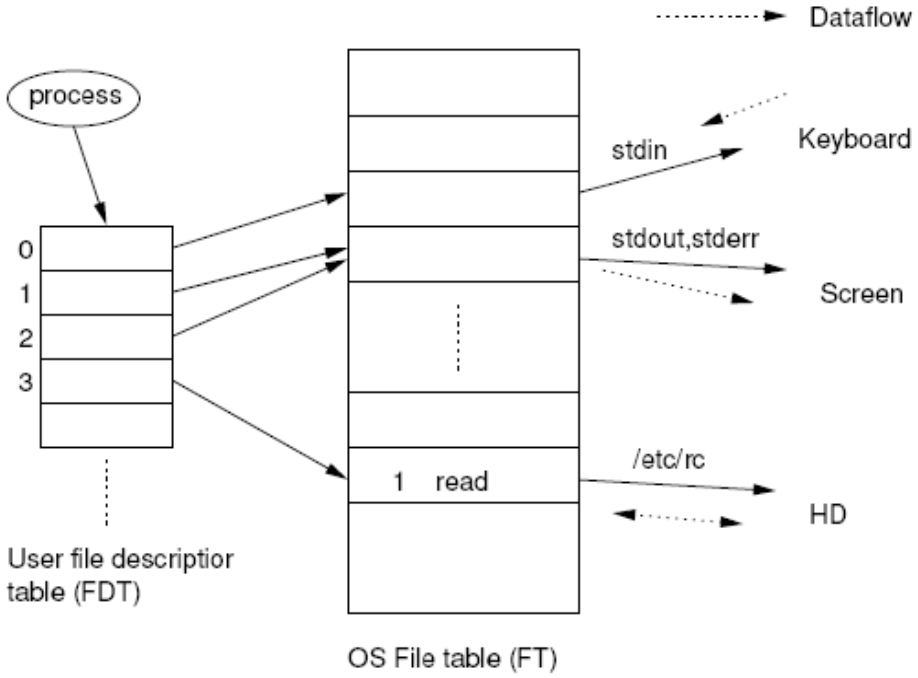
Figur. Datastrukturerna efter lyckat anrop av `pipe(fd)`.

Den randiga saken till höger är pipen, som alltså är en kö, (FIFO-buffert) som ligger utanför processens minnesområde. Vi kommer åt den via fildeskriptorerna `fd[1]` och `fd[0]` där vi skriver till `fd[1]` för att lägga in något i kön och läser från `fd[0]` för att ta ut något ur kön.

I figuren ovan har vi inte tagit in någon extra process, vi illustrerar bara hur det ser ut när en process precis gjort pipe utan att koppla någon av läs eller skrivändarna mot någon annan process. För att hantera det här med andra processer behöver vi tala om *deskriptortabeller*.

Fildeskriptorer och systemanropet `open()`

Varje process kan ha ett antal filer öppna för läsning eller skrivning. En fil kan vara av många olika slag, det kan röra sig om en socket, en pipe, en vanlig fil på hårddisken osv. Alla dessa typer av objekt adminstreras i något som heter *fildeskriptorer*. En fildeskriptor är egentligen bara ett heltal som finns tillgängligt för processen i en för processen intern tabell som heter *User File Descriptor Table* (*FDT*). Denna tabell pekar in i operativsystemets tabell (*OS File Table*, *FT*) som förtecknar *alla* öppna filer i hela systemet. Det är via dessa tabeller som operativsystemet administrerar *IPC* (*InterProcess Communication*). Då en process öppnar en fil, till exempel `/etc/rc`, får filtabellerna följande utseende



Utseende av datastrukturerna efter öppning av filen `/etc/rc`.

Figuren visar hur datastrukturerna ser ut efter körning av systemanropet `fd = open("/etc/rc",O_RDONLY);` En ny fildeskriptor har skapats med nummer 3. Fildeskriptorerna 0, 1 och 2 är tre fildeskriptorer som alla processer får öppna från start, det är `STDIN`, `STDOUT` och `STDERR`, alltså standardströmmarna för in- och utmatning samt felrapportering. Som default är dessa kopplade till tangentbordet (`STDIN`) respektive skärmen (`STDOUT` & `STDERR`). Då processen gör `open()` returneras nästa lediga fildeskriptor och det var som sagt i det här fallet 3. Ett nytt anrop till `open()` innan `fd` stängs skulle returnera 4. En process egna fildeskriptor tabell (User file descriptor table – FDT) håller alltså reda på vad 0, 1, 2, (alltså deskriptorindexena) betyder för varje enskild process. De kan alltså ha olika betydelse för olika processer. De pekar alltid in i den stora deskriptortabellen, OS file table (FT) där operativsystemet håller reda på samtliga öppna filer i systemet. Det är via den här tabellen som vi kan kommunicera mellan processer, antingen genom pipes eller sockets, lokalt på samma dator eller över nätet.

Systemanropet `close(2)`

Ett anrop av `close()` ska göras med en parameter som anger index för den post i FDT som ska stängas. Vid systemanropet `close()` frigörs motsvarande post i FDT och även i FT under förutsättning att ingen annan deskriptor refererar till samma post.

PI-Fråga:

Vad händer med FDT vid ett `fork()`-anrop?

På den förberedande övningen undersökte vi en konstellation med två pipes, en föräldra- och en barnprocess som kommunicerade via två pipes, en som hette `request` och en som hette `response`. Programmet med två pipes exemplifierade en vanlig situation där en process producerar data (föräldern) och en annan process konsumerar data (barnet). Vi ska nu se hur fildeskriptortabellerna fungerar tillsammans med `fork()`-anropet.

Om en process anropar `fork()` kommer det att skapas en barnprocess med sin egen FDT som är en kopia av föräldraprocessens FDT. I detta fall gäller att *alla deskriptorer pekar till samma post i den gemensamma OS FT och de delar på samma offset (alltså hur långt man läst/skrivit i filen)*. Här finner vi förklaringen till varför barnprocess och föräldraprocess som delar på en pipe måste göra stängningar av sina respektive ändar som de inte ska använda innan det egentliga arbetet kan ta vid:

Barnet gör:

```
close(READ);                 (Stänger alltså sin STDIN.)
close(WRITE);                (Stänger alltså sin STDOUT.)
close(request[WRITE]);       (Stänger alltså sin skrivände av request-pipen.)
close(response[READ]);       (Stänger alltså sin läsände av response-pipen.)
```

Föräldern gör:

```
close(request[READ]);        (Stänger alltså sin läsände av request-pipen.)
close(response[WRITE]);      (Stänger alltså sin skrivände av response-pipen.)
```

Sedan kan kommunikationen starta. (Föräldern behåller sina `STDIN` och `STDOUT` öppna för kommunikation med omvärlden.)

Vi kan nu noggrannare beskriva systemanropen `read()` och `write()` som ligger till grund för all in- och utmatning i ett *UNIX*-system. På förra kursmötet körde vi `read(0,...)` respektive `write(1,...)` och det kanske inte var helt klart varför vi hade 0 i anropet till `read()` och 1 i anropet till `write()`. Nu kan vi dock förklara detta genom att nämna att nollan och ettan är *fildeskriptorer* och vi kan nu ge den fullständiga beskrivningen av de båda systemanropen `read()` och `write()`. De bygger alltså på att vi har fildeskriptorer för den process som anropar `read()` och `write()`.

Systemanropet `read(2)`

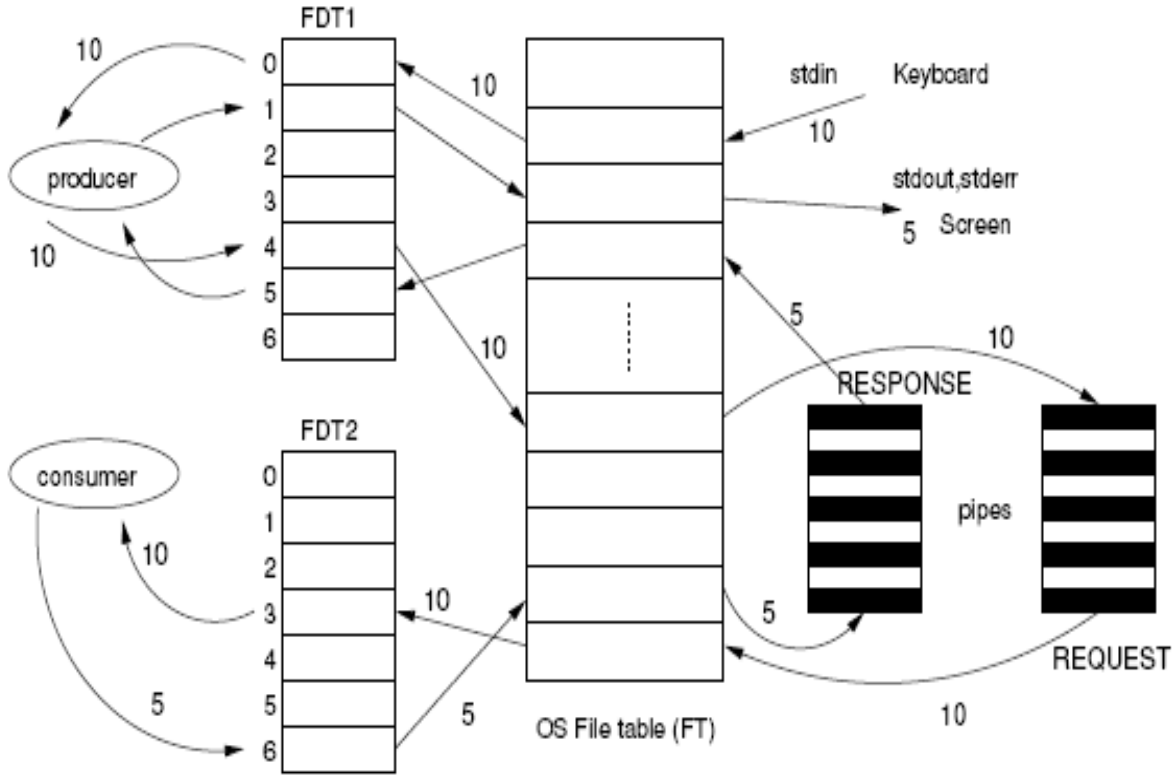
Med hjälp av detta systemanrop kan man läsa in data från en fildeskriptor. Detta är ofta `STDIN`, då läsning per default sker från tangentbordet, men en process kan också få data matade till sig. Själva inplaceringen av datana sker med `read()`. Vid anropet lämnas tre parametrar. Den första är ett heltalsindex som anger vilken post i FDT som ska användas för att peka ut den fil man arbetar mot. Man skickar alltså in fildeskriptorn här. Den andra parametern anger en buffert till vilken data ska kopieras vid läsning och den tredje anger hur många bytes som ska läsas. Observera att man kan läsa in data från `STDIN` genom att ange fildeskriptorindex till 0 och att vanliga `scanf()`, och liknande, läser med `read()` från `STDIN`. Efter lyckad genomförd läsning anger returvärdet hur många bytes som verkligen lästes. En misslyckad läsning ger som vanligt returvärdet – 1. Då värdet som returneras är 0 indikerar detta filslut. (Läs `man 2 read`.)

Systemanropet `write(2)`

Med hjälp av detta systemanrop kan man skriva ut data till en fil. Vanligtvis sker detta via `STDOUT` eller `STDERR` som är fildeskriptorerna för utmatning på skärmen. Vid anrop lämnas tre parametrar. Den första parametern är ett heltalsindex som, på samma sätt som för `read()`, anger vilken post i FDT som ska användas för att peka ut den fil man arbetar mot. På samma sätt som för `read()` skickar vi allstå in fildeskriptorn här. Den andra parametern anger en buffert från vilken data ska kopieras vid skrivning och den tredje anger hur många bytes som ska skrivas. Observera att man kan skriva ut data till `STDOUT` eller `STDERR` genom att ange fildeskriptorindex 1 eller 2 och att vanliga `printf()`, och liknande, skriver med `write()` till `STDOUT`. Efter lyckad genomförd skrivning anger returvärdet från `write()` det antal bytes som skrivits. Även här ger en misslyckad läsning som vanligt returvärdet – 1.

Fullständig karta över övningen

Med dessa förklaringar kan vi nu också ge en fullständig beskrivning av hur övningen fungerar. Nedan visas hur de olika piparna och processerna i övningen ser ut i relation till sina respektive fildeskriptortabeller och operativsystemets fildeskriptortabell. Det är som sagt två processer, en producent (föräldern) och en konsument (barnet) (*Producer* och *Consumer*) som använder två pipes `REQUEST` och `RESPOND`, för kommunikation. Producentprocessen läser från `STDIN` ett tal som användaren skriver in på tangentbordet, t ex 10. Detta tal skickas till `REQUEST`-pipens skrivöppning. Konsument-processen läser talet 10 från `REQUEST`-pipens läsöppning och beräknar hur många primtal som finns upp till och med det angivna talet 10, det blir 4 stycken (2, 3, 5 och 7) och skriver in 5 på `RESPOND`-pipens skrivöppning. Producenten läser talet 4 från `RESPOND`-pipens läsöppning och skriver resultatet till `STDOUT`. I övning 5 finns den programkod som motsvarar processerna i figuren nedan.



Figur. De två processerna från övningen som kommunicerar med två pipes.

I det första anropet till `pipe()`, alltså "`pipe(request);` ", tilldelas `request[0]` värdet 3 och `request[1]` tilldelas värdet 4 för de är de två första lediga fildeskriptorerna (efter 0, 1 och 2 som redan är tilldelade `STDIN`, `STOUT` respektive `STDERR`.) I nästa `pipe()`, som skapar `RESPOND`-pipen, används fildeskriptorerna 5 och 6. Efter `fork()` blir barnprocessen *Consumer* (alltså konsumentprocess) och föräldraprocessen blir *Producer* (alltså producentprocess.) Barnprocessen har som `fork()`-ad process, en kopia av föräldraprocessens FDT. Båda processernas FDT pekar på OS FT och således delar föräldra- och barnprocessen på båda pipar. Producenten använder endast skrivöppningen på `REQUEST`-pipen och läsöppningen på `RESPOND`-pipen och stänger därför läsöppningen på `REQUEST`-pipen och skrivöppningen på `RESPOND`-pipen med kommandot "`close(request[READ]);`" samt "`close(respond[WRITE]);`". Här är `READ` och `WRITE` fördefinierade konstanter med värdena 0 respektive 1. Konsumentprocessen, det vill säga barnprocessen, stänger i sin tur skrivöppningen på `REQUEST`-pipen samt läsöppningen på `RESPOND`-pipen. Barnprocessen stänger även sina kopior av `STDIN` och `STDOUT`, vi kan se det i satserna "`close(READ);`" respektive "`close(WRITE);`".

Frågor att fundera över (Vi gör inte Peer Instruction här, men vi pratar kring dessa saker):

* Studera koden i kommunikationsövningen, alltså avsnittet

```
close(READ);                (Stänger alltså sin STDIN.)
close(WRITE);               (Stänger alltså sin STDOUT.)
close(request[WRITE]);      (Stänger alltså sin skrivände av request-pipen.)
close(response[READ]);      (Stänger alltså sin läsände av response-pipen.)
```

Är det helt nödvändigt att man gör så många stängningar? Vad ska dessa stängningar tjäna till? Vad händer om man utelämnar någon av dem?

* Hur möjliggör stängningarna att programmet kan fullborda sin körning? (*Ledning:* Fundera över vad som får `while`-loopen att fullbordas.)

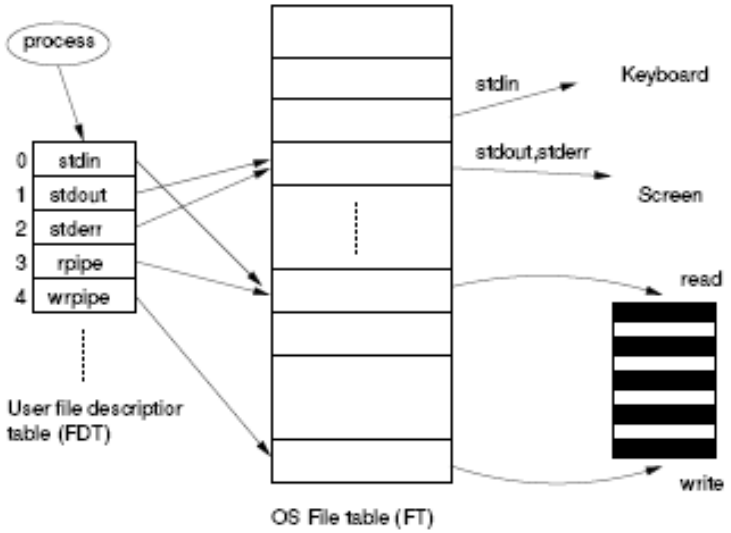
* Vad händer om en av processerna plötsligt dör? Kan den andra processen upptäcka det?

Systemanropet `dup()`

Nu har vi lite förståelse om hur processer kan kommunicera via pipes och hur dessa är arrangerade i processers fildeskriptortabeller och hur dessa relaterar till operativsystemets fildeskriptortabell. Vi ska nu se hur vi gör omdirigeringar genom att ändra om i processers FDT. Vi har sett hur vi kan ta bort en association till OS File Table genom `close()`. Nu ska vi se på `dup()` som verkar åt andra hållet: med `dup()` skapar vi flera associationer till OS File Table. Genom att kombinera `dup()` med `fork()` och `exec()` ska vi sedan se hur vi kan ta andra program (som vi inte skrivit) och koppla ihop dem i *C,* vi ska se på *C*:s motsvarighet till hur man åstadkommer en manöver av typen `$ kommando | kommando`. (`$` ska motsvara en bashprompt.)

Systemanropet `dup(fd)` duplicerar (kopierar) en befintlig fildeskriptor `fd` så att en process kan tillgå en och samma fil eller pipe via två olika fildeskriptorer. Kopieringen sker till första lediga fildeskriptor i FDT, det vill säga den med lägst index, och lämnar vid lyckat anrop som returnvärde detta index. En fildeskriptor i FDT lediggörs genom `close(fildeskriptorn);`. I figur 6 visas exempel på en datastruktur i `FDT` efter ett lyckat systemanrop med `close(0);` följt av `dup(3);`. Av figuren kan vi se att processen kommer att läsa från pipens läsöppning istället för

`STDIN`, alltså tangentbordet. Om man gör omdirigering med `close()` och `dup()` på detta sätt och sedan gör en läsning med till exempel `scanf()` så kommer läsningen att ske från pipens läsända och inte från tangentbordet. Läsningen har, som sagt, dirigerats om.



Figur. Datastrukturerna i FDT och FT efter close(0); och dup(3);.

Deskriptorerna `rpipe` och `wrpipe` är 3 respektive 4 som indikerar, i FDT, associationer till pipen. Men efter dessa manövrar kan vi alltså läsa från pipen från strömmen STDIN, som har index 0. Här är det då ofta lämpligt att stänga `rpipe` (som är 3) för att bara ha en association till läsänden av pipen.

PI-fråga:

Att koppla ihop `fork()`, `exec()` och `pipe()`

Det övergripande målet med den stora kommunikationsövningen (förbereder oss för laboration 1) är att koppla ihop tre processer som startats med `fork()`/`exec()` och låta dessa kommunicera med pipes. Detta kommer att illustrera en viktig allmän princip som *UNIX*/*POSIX* bygger på: att kunna sätta ihop flera mindre program, som vart och ett uför små uppgifter, till ett större program som utför en mer komplicerad uppgift. *UNIX*/*POSIX* filosofi är som programmerarens:. sätt ihop flera små väl fungerande enheter till en stor väl fungerande enhet. De processer som vi ska sätta ihop kommer att vara vanliga enkla *UNIX*/*POSIX*-kommandon, vi ska i *C* göra ett anrop av typen

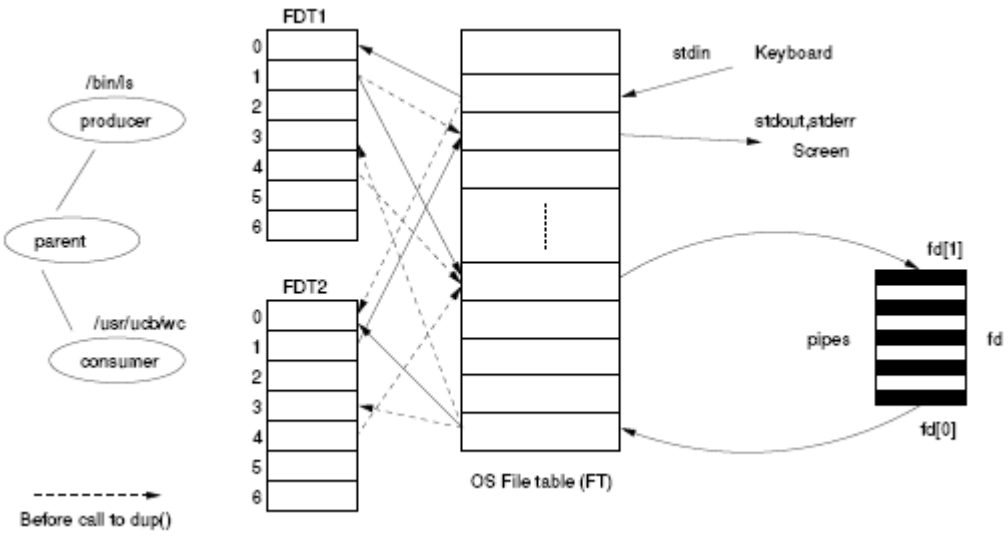
```
> ls -l /bin/?? | grep rwxr-xr-x | sort
```

Detta anrop tar fram alla körbara kommandon som består av två bokstäver med rättighetsprofiler `rwxr-xr-x` och sorterar dessa.

För att kunna koppla ihop processer som inte ligger i samma körbara fil (som till exempel kommandona `ls`, `grep` och `sort` ovan) så måste vi meka med deras STDIN och STDOUT. För att lyckas med det ska vi först studera fildeskriptorer mer noggrannt.

Ovan (och i övningen) har vi programmerat parallella processer och låtit dem kommunicera med hjälp av pipes som kopplats ihop med fildeskriptorer som vi deklarerat i det program vi skrivit. Vi har sett att det var viktigt att stänga alla ändar på pipen som man inte använder. Det kommer att vara viktigt även i fortsättningen. Vi ska nu se mer fördjupat på de bakomliggande datastrukturerna som operativsystemet tillhandahåller runt pipehantering. Detta ska leda in på hur `STDIN` och `STDOUT` kan dirigeras om för att möjliggöra den ihopkoppling som vi eftersträvar i den stora kommunikationsövningen.

Programexempel som visar omdirigeringar
I figuren nedan ser vi en mer avancerad användning av pipes för kommunikation. En föräldraprocess har skapat två barnprocesser som kommunicerar via en pipe vars läsöppning och skrivöppning har dirigerats om till barnprocessernas STDOUT respektive STDIN. Figur 8 visar koden för hur detta ska implementeras.



Figur. En mer avancerad användning av pipes.

Programmet:

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

char p1[]="/bin/ls", p2[]="/usr/bin/wc";      //Globala variabler. Fy!
main() {
  int status, fd[2], test; printf("%s | %s \n", p1, p2);
  if (pipe(fd)<0) {
    perror("Can't create pipe.\n");
    exit(1);
  }

  switch(fork()) {
    case 0: /* Process 1 - The producer*/
      test = close(WRITE); assert (test==0);
      test = dup(fd[WRITE]); assert (test==WRITE);
      test = close(fd[READ]); assert (test==0);
      test = close(fd[WRITE]); assert (test==0);
      execlp(p1, p1, (char*)0);
      perror("Cannot execl 1.");
      exit(1);
    case -1:
      perror("Cannot fork.");
      exit(1);
  }
  switch(fork()) {
    case 0: /* Process 2 - The consumer */
      test = close (READ); assert (test==0);
      test = dup (fd[READ]); assert (test==READ);
      test = close (fd[WRITE]); assert (test==0);
      test = close (fd[READ]); assert (test==0);
      execlp(p2, p2, (char*) 0);
      perror("Cannot execl 2.");
      exit(1);
    case -1:
      perror("Cannot fork.");
      exit(1);
  }

  test = close(fd[READ]); assert(test==0);
  test = close(fd[WRITE]); assert (test==0);
  wait(&status);
  wait(&status);
```

```
}
```

Det här sista programmet är nyckeln till den stora övningen på processkommunikation. Vi ska gå igenom programmet och förklara.

1. Först skapas en pipe: läsöppning `fd[0]`, skrivöppning `fd[1]`.

2. Två systemanrop till `fork()` görs för att skapa de två barnprocesserna..

3. Vi använder `execlp()` för att förvandla den översta barnprocessen till `ls`. System-anropet `execlp()` utplånar alla spår av den process som anropar `execlp()` och barnet kommer således att fullständigt förvandlas till processen `ls`. **Innan detta sker ställs dock fildeskriptorerna om så att barnet (som ska bli `ls`) skickar data på pipens skrivöppning.** Vanliga `ls` ger ju ut en lista på filer i nuvarande katalog och listan kommer ut på `STDOUT`. Vad vi gör med `close()` och `dup()` är alltså att dirigera om detta flöde till pipens skrivöppning. Observera noga alla stängningar av de övriga ingångarna till olika fildeskriptorer.

4. Vi använder även `execlp()` för att förvandla den undre barnprocessen till en annan process, i det här fallet `wc` som räknar antalet ord/rader/tecken som den får in på `STDIN`. Vi ställer om barnets `STDIN` med `dup()` och `close()` så att barnet läser från läsöppningen av pipen och anropar sedan `execlp()`. Barnprocessen blir då processen `wc` kommer då att läsa från läsöppningen av pipen och således har vi kopplat ihop `ls` med `wc` i två barnprocesser.

5. Programmet vimlar av `assert`-satser. Dessa används i samband med testning och felsökning för att bekräfta att vissa händelser går att genomföra. Exekvering av `assert(<villkor>);` där villkoret evaluerar till ett sant värde, fungerar och programmet kan fortsätta köra. Då vi kör assert på ett falskt värde avbryts däremot exekveringen och vi får ett felmeddelande om vilken rad som misslyckats. Satserna behövs inte då programmet fungerar men är mycket bra att använda då vi utvecklar och avlusar programmet. Observera att man kan ange att `assert`-satser inte behöver tas med i kompilering, man kan ange till kompilatorn att de ska ignoreras, de är då viktigt att man inte baserar någon körning på att de exekveras, således är det bättre att skriva

```
test=close(WRITE); assert(test==0);
```

än

```
assert(close(WRITE)==0);
```

Nu är det definitivt läge att sätta igång med den stora kommunikationsövningen. Den kommer att vara mycket bra inför laboration 1.