KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Institutionen för Mikroelektronik och
Informationsteknik

Användbara systemanrop och biblioteksfunktioner

2G1520 Operativsystem

2005

Institutionen för Mikroelektronik och
Informationsteknik

1.0 Introduktion

Det här häftet beskriver några systemanrop och biblioteksfunktioner du kommer att ha användning av i de första laborationerna i Operativsystemskursen. Läs häftet innan du börjar med laborationerna och fundera på de frågor som finns i texten.

De funktioner som häftet omfattar är grundläggande funktioner för: processhantering, signalering, filhantering och kommunikation via pipes. Du kommer att behöva läsa motsvarande manualsidor för dessa funktioner också, men förhoppningsvis skall beskrivningarna här göra det enklare att förstå manualsidorna.

2.0 Processer

I de flesta operativsystem, och speciellt i Unix, har varje process ett unikt **p**rocess**id**entifikationsnummer som kallas PID (Process ID).

2.1 Systemanropet fork(2)

En ny UNIX-process skapas med hjälp av systemanropet `fork`. Systemanropet `fork` skapar en nästan exakt kopia av den process som anropade fork, men med ett nytt unikt PID.

Processen som anropar `fork` kallas för förälder eller *parent-process* och den nya processen kallas för barn eller *child-process*.

Efter genomförd `fork`-operation har både parent- och child-processen samma programkod. Båda processerna kommer, efter retur från systemanropet `fork`, att fortsätta exekveringen på samma plats i programmet, nämligen i programkoden omedelbart efter anropet av `fork`.

För att kunna skilja de två processerna levererar systemanropet `fork` vid returen olika returvärden till parent- och child-processerna. Värdet 0 lämnas som returvärde till child-processen medan parent-processen får ett returvärde som utgörs av PID för den nya processen, alltså child-processens PID.

En vanlig sekvens för att skapa en ny process visas i Figur 1. nedan.

Studera manualtext för `fork` *i manualsektion 2.*
Kommando: `man -s 2 fork` *(SysV),*
`man 2 fork` *(BSD).*

2.2 Systemanropen getpid(2) och getppid(2)

Studera manualtexten för getpid-anropen.
Kommando: `man -s 2 getpid` *(SysV),*
`man 2 getpid` *(BSD).*

```
/* Enkelt exempel på användning av fork() */
int pid;

pid = fork();
if ( pid == 0 )
{
  /* Denna kod körs endast i child-processen */
}
else
{
  /* Denna kod körs endast i parent-processen */
  if( pid == -1 )
  {
    /* ERROR */
  }
  else
  {
    /* Denna kod körs i parent och endast om fork lyckades */
  }
}
```

Figur 1. Systemanropet fork()

Om en process vill ha reda på sitt eget PID används systemanropet `getpid` som returnerar den anropande processens PID.

Om en process vill ta reda på PID för sin parent-process används systemanropet `getppid` som returnerar PID för parent-processen.

2.3 Systemanropet exec()

För att kunna starta exekvering av en process med annan kod än parent-processen används ett systemanrop av typen `exec`.

Studera manualtexten för exec-anropen.
Kommando `man -s 2 exec` *(SysV),*
`man 2 exec` *(BSD). Om ingetdera fungerar, prova manualsektion 3 i stället.*

Vid varje systemanrop med någon variant av `exec` ska parametrar levereras. Första parametern anger vilken fil som ska exekveras i stället för det program som anropar `exec`. Dessutom kan man ge ett antal parametrar till programmet i filen och därvid använda olika metoder för parameteröverföring.

Om `exec`-anropet lyckas så försvinner det gamla programmet som kördes i processen. Programmet ersätts av ett nytt program, med nya data och nya kommandoradsparametrar.

Om `exec`-anropet misslyckas görs retur med returvärdet -1, till den anropande processen som då förväntas vidtaga lämpliga åtgärder, antingen rätta till felen och göra nytt försök eller ge larm.

Man kan se en process som en påse i vilken man placerar ett program medan det körs. Systemanropet `exec` byter ut innehållet (programmet) i påsen, men påsen själv (processen) finns kvar och har samma process-ID (PID) som före exec.

Applicerar man liknelsen med påsen på `fork`, så kommer `fork` att skapa en ny påse (process) som är precis likadan som den gamla, med samma innehåll (program) och allt. Bara PID skiljer (och så returvärdet från `fork`-anropet).

2.4 Utförligt kodexempel

Här följer ett utförligt exempel på användning av `fork` och `exec`. Se Figur 2.. Programkoden visas med fetstil, kommentarerna med tunnare bokstäver.

I början av programmet inkluderas ett antal filer med `#include`. De inkluderade filerna finns i ett standardbibliotek, `/usr/include`, och innehåller definitioner av systemanrop med mera.

Filen `types.h` definierar nya variabeltyper för värden som används i systemet. Det kan vara så att ett process-ID inte får plats i en `int` på det Unix-system där programmet körs; därför används en särskild typ `pid_t` i stället. Filen types.h definierar `pid_t` till att vara `int`, `long int` eller något annat lämpligt.

`char * errormessage` deklarerar en variabel som kan innehålla en pekare till `char`, alltså en pekare till bokstäver och tecken.

Skrivsättet `if( 0 == variabel )` används för att minska risken för den farliga felskrivningen `if( variabel = 0 )`, som medför att variabeln tilldelas värdet 0 inuti if-villkoret. Skulle man glömma ett av likhetstecknen så att `if( 0 == variabel )` i stället råkar bli `if( 0 = variabel )` så kommer kompilatorn att klaga. Eftersom konstanten 0 inte kan tilldelas något nytt värde så är uttrycket `0 = variabel` ogiltigt.

2.5 Varianten execl(3)

Av de olika möjliga varianterna på exec använder vi här `execl`. Parametrarna till `execl` är en lista med pekare till textsträngar.

Den första parametern till `execl` anger sökvägen till det program som ska köras, i detta fall `/bin/echo`. Programmet `echo` tar en lista med parametrar och skriver ut dem.

De parametrar som skickas till programmet echo kommer efter den inledande `"/bin/echo"`. Programmet `echo` kommer alltså att få tre parametrar: `"/bin/echo"`, `"Hello, world!"` och 0 (som betyder slut på parameterlistan).

Ett Unix-program ska alltid få sitt eget programnamn som allra första parameter. Några få program använder den, övriga kastar bort den. Men den måste alltid finnas.

Den första upplagan av "`/bin/echo`" i `execl`-anropet används av `execl` själv för att hitta programmet. Allt annat skickas till programmet `echo`, och `echo` förväntar sig att få veta hur den anropades. Därför finns "`/bin/echo`" en gång till, och denna upplaga kommer till programmet `echo`.

2.6 De olika exec-varianterna

Systemanropet exec finns i olika varianter med bokstäverna `e`, `l`, `p` och `v` efter `exec`.

Bokstäverna `l` och `v` står för lista respektive vektor och anger hur argumenten till programmet ska anges. I detta lab-PM har vi bara visat varianten `l` där argumenten listas i anropet.

Varianten `v` kräver att programmet fyller i en `argv`-vektor före anropet. Det är ett krångligare sätt, men det klarar godtyckligt antal argument.

```
/* forktest.c - vanlig användning av systemanropet fork() */

/* Include-rader ska vara allra först i ett C-program */
#include <sys/types.h> /* definierar bland annat typen pid_t */
#include <errno.h> /* definierar felkontrollvariabeln errno */
#include <stdio.h> /* definierar stderr, dit felmeddelanden skrivs */
#include <stdlib.h> /* definierar bland annat exit() */
#include <unistd.h> /* definierar bland annat fork() */

pid_t childpid; /* processid som returneras från fork */

int main()
{
  /* Följande kod ska finnas inne i main() eller annan procedur! */

  childpid = fork();
  if( 0 == childpid )
  {
    /* denna kod körs endast i child-processen */
  }
  else
  {
    /* denna kod körs endast i parent-processen */

    if( -1 == childpid ) /* fork() misslyckades */
      {
        /* i C får man alltid deklarera variabler efter vänster krullparentes */
        char * errormessage = "UNKNOWN"; /* felmeddelandetext */

        if( EAGAIN == errno )  errormessage = "cannot allocate page table";
        if( ENOMEM == errno ) errormessage = "cannot allocate kernel data";
        fprintf( stderr, "fork() failed because: %s\n", errormessage );
        exit( 1 ); /* programmet avbryts om detta fel uppstår */
      }

    /*
      Kommer vi hit i koden så är vi i parent-processen
      och fork() har fungerat bra - i så fall innehåller
      variabeln childpid child-processens process-ID
    */

  }

  exit( 0 ); /* programmet avslutas normalt */
}
```

Figur 2. Systemanropen fork() och execl()

I varianter med `p` söker exec igenom katalogerna som anges i environment-variabeln PATH för att hitta programmet som ska köras. Utan `p` måste hela sökvägen anges, till exempel "`/bin/ls`".

Varianter med `e` gör det möjligt att ange programmets environment vid anropet.

2.7 Felhantering

Om det inte går att utföra fork av någon anledning så sätts systemvariabeln `errno` till en felkod. Felkoden är ett heltal som anger vad som gått fel.

I Unix-system finns ett antal olika felkoder definierade, med namn som används i stället för numren. I manual-texten för execve får du veta vilka som finns i ditt system.

För felutskrifter används den speciella utmatningsströmmen standard error, som skrivs `stderr`. Det finns mer att läsa om detta senare i detta lab-PM. `fprintf` är en `printf`-variant där man anger vilken utström utmatningen ska till.

Systemanropet `exit` avslutar en process. Parametern till exit används för felkontroll. Ett program som avslutas på normalt sätt ska göra `exit( 0 )`. Alla andra värden anger att ett fel har uppstått.

3.0 Processkommuniktion med signaler

Unix-processer kan kommunicera genom att skicka signaler till varandra för att meddela att olika händelser inträffat. Det finns olika signaler med olika namn och nummer.

Titta på manualtexten för `signal`, den ger en bra översikt över vad signaler är och hur de används i Unix.

Att en process tar emot en inkommande signal kan liknas vid ett avbrott som orsakats av mjukvaran, ett mjukvaru-avbrott. Avbrott = interrupt, mjukvaruavbrott = software interrupt.

Om en process under sin exekvering tar emot en signal, så kommer processen att avbrytas och operativsystemet måste se till att en speciell avbrottsrutin exekveras.

Notera att *avbrottsrutin* på engelska heter *interrupt handler,* vilket i detta sammanhang blir *signal handler,* som i sin tur försvenskas till *signalhanterare*.

Efter det att signalhanteraren har exekverats sker returhopp till den process som avbröts. Exekveringen forsätter där på det ställe där den avbröts. För varje process ska det finnas en särskild signalhanterare för varje enskild signal som kan uppträda. Operativsystemet har signalhanterare som används om processen inte installerar egna.

Man kan jämföra detta med hantering av exceptions i en Motorola 68000-processor som använder vektoriserat avbrott. I en avbrottsvektortabell finns, för varje exception, en adress till den avbrottsrutin, exception handler, som ska anropas vid motsvarande exception.

På motsvarande sätt har varje process sin egen vektortabell för att hantera varje tänkbar inkommande signal.

Vid användning av avbrottsrutiner i Motorola 68000-processorn måste programmen initiera avbrottsvektortabellen genom att skriva in startadressen till den avbrottsrutin som man vill ska anropas för ett speciellt avbrott eller exception.

På motsvarande sätt måste varje process initiera sin egen vektortabell för signaler genom att ange vilken signalhanterare som ska anropas för en signal av en viss typ. Detta kallas att *installera* signalhanteraren.

Operativsystemet har standardrutiner (default signal handlers) för signaler som processen inte installerat egna signalhanterare för.

3.1 Funktionen sigaction(2)

Initiering av signalvektortabellen kan göras med systemanropet `sigaction`.

Studera manualtext för systemanropet `sigaction`.

Med systemanropet `sigaction` anges vilket beteende processen ska ha för varje signal.

Ofta anger man en rutin, signalhanterare, som ska ta hand om och fånga upp en viss signal. Detta kallas för att installera signalhanteraren och motsvarar initiering av avbrottsvektortabellen i en Motorola 68000-processor. Det går också att ange att en viss signal ska ignoreras.

Det finns flera olika biblioteksfunktioner för att sköta signalhantering, till exempel `signal` och `sigset`. Dessa ingår inte i Posix-standarden, och fungerar på olika sätt beroende på vilken Unix-variant som används.

3.2 System-filen signal.h

Det finns ett antal signaler definierade i Unix. I C är namnet på signalen, till exempel `SIGINT`, definierat som ett makro som kan skrivas i stället för motsvarande signalnummer.

Varje signal har ett namn (makro i C), till exempel `SIGKILL` och `SIGINT`, som motsvaras av heltalsvärden. Så har `SIGKILL` till exempel värdet 9 och `SIGINT` värdet 2.

Makrodefinitionerna tas med vid kompileringen om filen `/usr/include/signal.h` inkluderas. Men definitionerna finns inte i själva filen `/usr/include/signal.h`. I stället innehåller denna fil en ny `#include`-sats, och definitionerna finns alltså i en annan fil.

Leta upp makrodefinitionerna. Läs och fundera över dem! Tips: är ditt Unix-system SunOS 5 så titta i `/usr/include/sys/signal.h`; kör du Red Hat Linux så kan definitionerna finnas i filen `/usr/include/asm/signal.h`.

3.3 Systemanropet kill (2)

Studera manualtexten för systemanropet `kill`. Kommando: `man -s 2 kill` (SysV), `man 2 kill` (BSD). Observera att "`man kill`" ger information om användarkommandot kill, som är något annat.

För att skicka en signal från en process till en annan process används systemanropet `kill(2)`. Ett anrop till `kill` ska ha två argument, nämligen PID och signalnummer.

Observera att man använder systemanropet `kill` och inte `signal` för att skicka en signal från en process till en annan process.

Systemanropet kill används bland annat:

- av en process för att signalera något till en annan process, eller grupp av processer;
- av Unix-kärnan för att signalera till någon eller några processer;
- av användare via kontrolltecken (till exempel Control-C, Control-Z);
- av kommandotolken (shell) för att implementera shell-kommandot `kill(1)`.

3.4 Programexempel med signalering mellan processer.

På nästa uppslag finns programkod som visar hur man kan använda signaler efter skapandet av en child-process med `fork`.

Funktionen `register_sighandler` fixar till de aningen krångliga parametrarna till systemanropet `sigaction`.

signal_handler är den signalhanterare som child-processen installerar för att ta emot, "fånga", signalen SIGINT.

På liknande sätt definieras i signalhanteraren cleanup_handler vad parent-processen gör då en `SIGINT` tas emot av parent-processen.

```
/* signaltest.c - användning av signaler */

#include <sys/types.h> /* definierar bland annat typen pid_t */
#include <errno.h> /* definierar felkontrollvariabeln errno */
#include <signal.h> /* deinierar signalnamn med mera */
#include <stdio.h> /* definierar stderr, dit felmeddelanden skrivs */
#include <stdlib.h> /* definierar bland annat exit() */
#include <unistd.h> /* definierar bland annat sleep() */

#define  TRUE ( 1 ) /* definierar den Boolska konstanten TRUE */

pid_t childpid; /* processid som returneras från fork */

void register_sighandler( int signal_code, void (*handler)(int sig) )
{
  int return_value; /* för returvärden från systemanrop */

  /* datastruktur för parametrar till systemanropet sigaction */
  struct sigaction signal_parameters;

  /*
   * ange parametrar för anrop till sigaction
   * sa_handler = den rutin som ska köras om signal ges
   * sa_mask    = mängd av övriga signaler som ska spärras
   *              medan handlern körs (här: inga alls)
   * sa_flags   = extravillkor (här: inga alls)
   */
  signal_parameters.sa_handler = handler;
  sigemptyset( &signal_parameters.sa_mask ); /* skapar tom mängd */
  signal_parameters.sa_flags = 0;

  /* begär hantering av signal_code med ovanstående parametrar */
  return_value = sigaction( signal_code, &signal_parameters, (void *) 0 );

  if( -1 == return_value ) /* sigaction() misslyckades */
  { perror( "sigaction() failed" ); exit( 1 ); }
}

/* signal_handler skriver ut att processen fått en signal.
   Denna funktion registreras som handler för child-processen */
void signal_handler( int signal_code )
{
  char * signal_message = "UNKNOWN"; /* för signalnamnet */
  char * which_process = "UNKNOWN"; /* sätts till Parent eller Child */
  if( SIGINT == signal_code ) signal_message = "SIGINT";
  if( 0 == childpid ) which_process = "Child";
  if( childpid > 0 ) which_process = "Parent";
  fprintf( stderr, "%s process (pid %ld) caught signal no. %d (%s)\n",
           which_process, (long int) getpid(), signal_code, signal_message );
}

/* cleanup_handler dödar child-processen vid SIGINT
   Denna funktion registreras som handler för parent-processen */
void cleanup_handler( int signal_code )
{
  char * signal_message = "UNKNOWN"; /* för signalnamnet */
  char * which_process = "UNKNOWN"; /* sätts till Parent eller Child */
  if( SIGINT == signal_code ) signal_message = "SIGINT";
  if( 0 == childpid ) which_process = "Child";
  if( childpid > 0 ) which_process = "Parent";
  fprintf( stderr, "%s process (pid %ld) caught signal no. %d (%s)\n",
           which_process, (long int) getpid(), signal_code, signal_message );

  /* if we are parent and signal was SIGINT, then kill child */
  if( childpid > 0 && SIGINT == signal_code )
  {
    int return_value; /* för returvärden från systemanrop */
    fprintf( stderr, "Parent (pid %ld) will now kill child (pid %ld)\n",
             (long int) getpid(), (long int) childpid );
    return_value = kill( childpid, SIGKILL ); /* kill child process */
    if( -1 == return_value ) /* kill() misslyckades */
    { perror( "kill() failed" ); exit( 1 ); }

    exit( 0 ); /* normal successful completion */
  }
}
```

Figur 3. Program som använder signaler, del 1

```
int main()
{
  fprintf( stderr, "Parent (pid %ld) started\n", (long int) getpid() );

  childpid = fork();
  if( 0 == childpid )
  {
    /* denna kod körs endast i child-processen */

    fprintf( stderr, "Child (pid %ld) started\n", (long int) getpid() );

    /* installera signalhanterare för SIGINT */
    register_sighandler( SIGINT, signal_handler );

    while( TRUE ) ; /* do nothing, just loop forever */

  }
  else
  {
    /* denna kod körs endast i parent-processen */

    if( -1 == childpid ) /* fork() misslyckades */
    { perror( "fork() failed" ); exit( 1 ); }

    /*
      Kommer vi hit i koden så är vi i parent-processen
      och fork() har fungerat bra - i så fall innehåller
      variabeln childpid child-processens process-ID
    */

    /*
     * Gör en liten paus innan vi registrerar parent-processens
     * signal-handler för SIGINT. Om vi trycker control-C i pausen
     * så avbryts parent-processen och dör. Vad händer då med
     * child-processen? Prova!
     */

    sleep( 1 );

    register_sighandler( SIGINT, cleanup_handler );

    while( TRUE )
    {
      int return_value; /* för returvärden från systemanrop */

      fprintf( stderr,
               "Parent (pid %ld) sending SIGINT to child (pid %ld)\n",
               (long int) getpid() , (long int) childpid );

      /* skicka SIGINT till child-processen, om och om igen */
      return_value = kill( childpid, SIGINT );
      if( -1 == return_value ) /* kill() misslyckades */
      {
        fprintf( stderr,
                 "Parent (pid %ld) couldn't interrupt child (pid %ld)\n",
                 (long int) getpid(), (long int) childpid );
        perror( "kill() failed" );
        /* programmet fortsätter även om detta händer */
      }

      sleep(1); /* gör en liten paus mellan varje SIGINT --
                   vi ska inte stress-testa systemet */

    }
  }
}
```

Figur 4. Program som använder signaler, del 2

Om användaren trycker Control-C i terminalfönstret så genereras signalen `SIGINT`. Denna signal skickas till den process som startats från terminalfönstret, det vill säga parent-processen i vårt fall.

Huvudprogrammet `main` börjar med att med att presentera sig med utskrift av PID. Därefter används systemanropet `fork` för att skapa en child-process.

Denna child-process gör först en utskrift och sedan registreras den signal-handler som ska anropas och "fånga" eventuellt inkommande `SIGINT`-signaler. Därefter snurrar child-processen runt i en oändlig slinga.

Parent-processen börjar med att registrera sin egen signalhanterare som fångar upp användarens Control-C från tangentbordet. Sedan går parent-processen in i en slinga som skickar upprepade signaler av typen `SIGINT` till child-processen.

Varje gång child-processen får en `SIGINT`-signal avbryts exekveringen i child-processens slinga och signalhanteraren signal_handler anropas. Signalhanteraren `signal_handler` skriver ut att den tagit emot signalen, sedan görs returhopp tillbaka till slingan i child-processen.

Om användaren trycker Control-C medför det att shell-processen skickar en signal av typ `SIGINT` till parent-processen. Parent-processen avbryts då och anropar sin signalhanterare (för `SIGINT`) – det vill säga cleanup_handler. Denna rutin dödar child-processen genom att skicka en signal av typen `SIGKILL` till child-processen med hjälp av systemanropet `kill`. Därefter avslutas parent-processen med systemanropet `exit`.

Som vanligt kontrolleras alltid returvärdet från systemanrop. För att förenkla programmet används här standardrutinen `perror`, som läser av `errno` och skriver ut rätt felmeddelande på `stderr`.

Skriv in detta program i filen `signaltest.c`. *Kompilera det med kommandot*
`gcc -o signaltest signaltest.c`

Kör igång detta program med kommandot `./signaltest`. *Prova det på olika sätt. Tryck Control-C och se vad som händer.*

Kör igång programmet igen. Öppna ett extra terminalfönster och använd kommandot
`ps -el | grep signaltest` *(SysV) eller* `ps -aux | grep signaltest` *(BSD) för att undersöka vad som händer. Ta reda på child-processens PID (till exempel 4711).*

Prova att stoppa child-processen tillfälligt med `kill -STOP 4711` *(om nu child-processen hade PID 4711). Vad händer i det andra terminalfönstret? Gör också ett nytt ps-kommando och studera resultatet. Kör igång child-processen igen med* `kill -CONT 4711`.

Prova nu att stoppa child-processen permanent med `kill -KILL 4711`. *Vad visar ps-kommandot nu?*

I `signal_handler` *används systemanropet getpid för att ta reda på PID för child-processen. Kan man istället använda variabeln* `childpid`*? Om ja, gör ändringen och förklara vad som händer. Om inte, varför?*

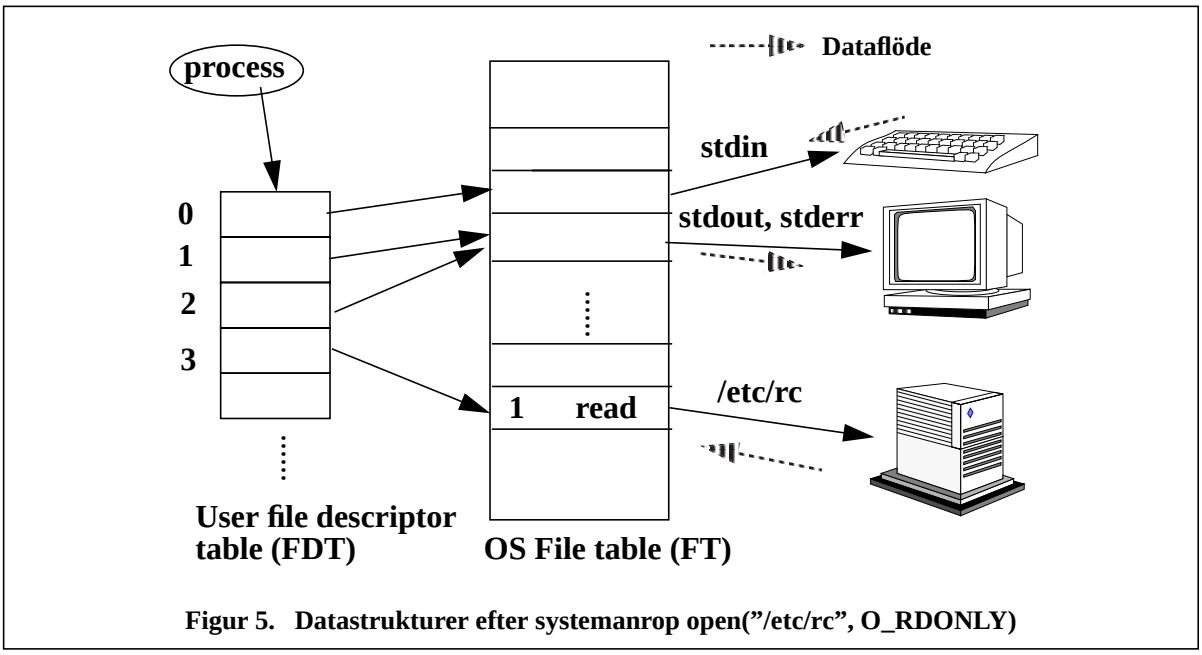
4.0 Processkommunikation via pipes

Ett vanligt sätt att kommunicera mellan processer är via filsystemet.

Vanliga systemanrop för in- och utmatning är `open(2)`, `read(2)`, `write(2)` och `close(2)`.

4.1 Systemanropet open(2)

Studera manualtext för systemanropet open. Kommando `man -s 2 open` *(SysV),*
`man 2 open` *(BSD).*



Figur 5. Datastrukturer efter systemanrop open(”/etc/rc”, O_RDONLY)

Systemanropet `open` returnerar index för en fildeskriptor i en användarprocess’ fildeskriptortabell (FDT), där en post i tabellen innehåller ett index till en post i operativsystemkärnans filtabell (FT).

I Figur 5. visas hur dessa datastrukturer hanteras i samband med att en process utför systemanropet `open(”/etc/rc”, O_RDONLY)`.

Ursprungligen har varje process som default tre öppna filer: standard input (`stdin`), standard output (`stdout`), och standard error (`stderr`) med fildeskriptorerna 0, 1 respektive 2.

Då processen exekverar `open` returneras **nästa lediga** fildeskriptor, i det här fallet 3. En post i FDT pekar ut (anger index för) en post i OS-kärnans FT som i sin tur pekar till den slutgiltiga filen. Returvärdet är alltså ett heltalsindex till en post i FDT.

Ett nytt anrop till `open` (innan man stängt filen ”`/etc/rc`”) skulle returnera **nästa lediga** fildeskriptor, det vill säga nummer 4 (även om man öppnar samma fil en gång till).

Fundera på hur pekarna i Figur 5. skulle se ut om man i en och samma process öppnar samma fil två gånger i rad. Om filen inte finns så skapas den, om processen har rättigheter att skapa filen. En post i FT (File Table) innehåller offset (avstånd) från början av filen samt filens mode (read, write). Offset ändras av systemanropen read och write.

4.2 Systemanropet close(2)

Ett systemanrop `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.

Studera manualtexte för systemanropet `close`.

4.3 Vad som händer med FDT vid ett fork-anrop.

Om en process anropar `fork` kommer det att skapas en child-process med en egen FDT som är en kopia av parent-processens FDT. I detta fall gäller att *alla deskriptorer pekar till samma post i den gemensamma File Table i operativsystemet och de delar på offset.*

Jämför hur FDT och FT hanteras i följande fall:

a) en process öppnar samma fil två gånger och läser via två fildeskriptorer;

b) en process gör `fork` *och både parent- och child-processerna läser parallellt;*

c) två godtyckliga processer öppnar och läser från samma fil.

4.4 Systemanropet read(2)

Med hjälp av detta systemanrop kan man läsa in data från en fil eller från `stdin`. Vid anropet lämnas tre parametrar. Den första parametern är ett heltalsindex som anger vilken post i File Descriptor Table som ska användas för att peka ut den fil man läser från. Den andra parametern är en pekare till en buffert som data ska kopieras till vid läsningen. Den tredje parametern anger hur många bytes som ska läsas, eller egentligen: det största antal bytes som man vill ska läsas. `read` kan läsa färre bytes än man begärde, till exempel vid slutet av en fil.

Observera att man kan läsa in data från stdin genom att ange fildeskriptor-index 0 och att "vanliga" `scanf` och liknande läser med `read` från `stdin`.

Efter en lyckad genomförd läsning anger returvärdet hur många bytes som verkligen lästes. En misslyckad läsning ger returvärdet -1.

Studera manualtext för systemanropet `read(2)`.

4.5 Systemanropet write(2)

Med hjälp av detta systemanrop kan man skriva ut data till en fil eller till `stdout` eller `stderr`. Vid anropet lämnas tre parametrar. Den första parametern är ett heltalsindex som anger vilken post i File Descriptor Table som ska användas för att peka ut den fil man skriver till. Den andra parametern är en pekare till en buffert från vilken data ska kopieras vid skrivningen.Den tredje parametern anger hur många bytes som ska skrivas.

Observera att man kan skriva ut data till `stdout` eller `stderr` genom att ange fildeskriptor-index till 1 eller 2 och att "vanliga" `printf` och liknande skriver med write till `stdout`.

Efter en lyckad skrivning anger returvärdet hur många bytes som verkligen har skrivits. En misslyckad skrivning ger returvärdet -1.

S*tudera manualtext för systemanropet* `write`.

4.6 Programexempel som visar användning av open, close, read och write.

I Figur 6. på nästa sida visas ett program som kopierar en fil. För att systemanropen ska synas tydligt så används tunnare bokstäver både för felhanteringskod och kommentarer.

Programmet hämtar namnen på ursprungsfil och kopia från kommandoraden. Denna information levereras till programmet som parametrar till `main`.Första parametern till `main` anger antalet argument. Eftersom du är insatt i hur `exec` fungerar så vet du att första argumentet till ett program ska vara filnamnet som användes för att starta programmet. Alltså ska ett program med två kommandoradsargument förvänta sig totalt 3 stycken argument: det egna programnamnet, sedan första argumentet från kommandoraden, och sist andra argumentet från kommandoraden. Andra parametern till main är en vektor med pekare till teckensträngar. Första strängen är alltså programnamnet, nästa sträng är första argumentet från kommandoraden och så vidare.

Programmet börjar med att öppna filen som skall kopieras för läsning. Skulle filen inte finnas, eller inte gå att öppna, så avslutas programmet. Om allt går bra så skapas den nya filen.

Kopieringen går till så att ursprungsfilen som skall kopieras läses ett block i taget och blocken skrivs till kopian vartefter de har lästs.

När hela ursprungsfilen har lästs kommer `read` att returnera värdet 0. Då stängs filerna med `close`, så att eventuella buffertar i operativsystemet töms ordentligt. Sedan avslutas programmet.

Programmet kontrollerar naturligtvis om något fel uppstår vid läsning/skrivning. I så fall skrivs ett felmeddelande ut, och programmet avslutas.

I exemplet i Figur 6. sker kopiering av samma information flera gånger. Först kopieras ett block med bytes från disken till en buffert i operativsystemskärnan. Därifrån kopieras blocket till programmets buffert i användarprocessen. Innehållet i denna buffert kopieras i sin tur till en ny buffert i operativsystemskärnan. Till sist kopieras innehållet i denna buffert i kärnan till den nya filen. Detta verkar ineffektivt. Kan man göra något åt det?

5.0 Rörledningssystem, pipes

Det finns fall då processer vill kommunicera tillfälligt, utan att explicit lagra resultatet av kommunikationen som mer eller mindre tillfälliga filer i filsystemet. I sådana fall är en tillfällig buffer utan namn en bra lösning. Systemfunktionen pipe uppfyller detta krav.

En pipe är en enkelriktad kanal för kommunikation mellan två eller flera processer. En pipe kan ses som en fifo-buffert och två fildeskriptorer: en fildeskriptor som anger ingång för skrivning, och en fildeskriptor som anger utgång för läsning. En eller flera processer kan läsa från/skriva till en pipe via systemanropen `read` och `write` med motsvarande fildeskriptorer för utgång respektive ingång till pipen.

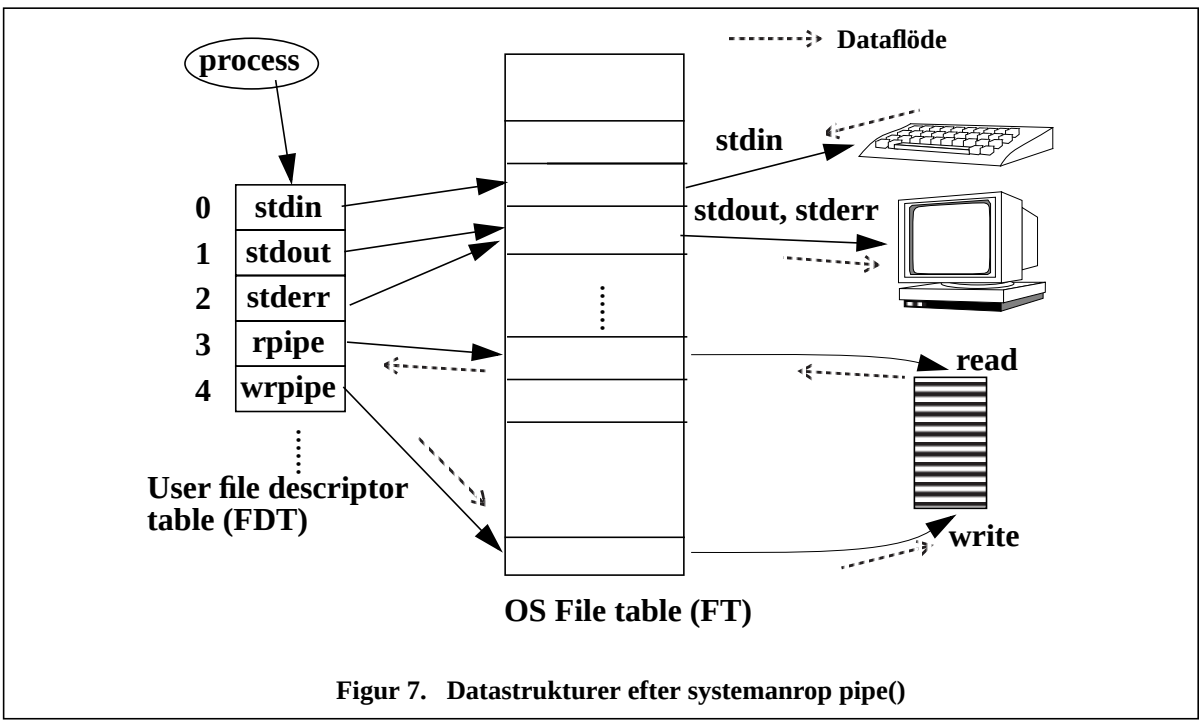
5.1 Systemanropet pipe(2)

För att allokera en pipe använder man systemanropet `pipe(2)`. Anrop till `pipe` ska ske med ett argument som är en heltalsvektor med två element (till exempel `int fd[2]`). Ett anrop av pipe resulterar i att två fildeskriptorer allokeras, den första, `fd[0]`, svarar mot utgången från pipen (läs-änden) och den andra, `fd[1]`, svarar mot ingången till pipen (skriv-änden).

Som returvärden levereras värdet 0 om anropet lyckades, och värdet -1 anger som vanligt ett misslyckande.

I Figur 7. visas exempel på File Descriptor Table och File Table efter ett systemanrop `pipe`.

Studera manualtext för systemanropet `pipe`.



Figur 7. Datastrukturer efter systemanrop pipe()

```c
/* filecopy.c - användning av open(), close(), read() och write() */
#include <sys/stat.h> /* definierar S_IREAD och S_IWRITE */
#include <errno.h> /* definierar errno och strerror() */
#include <fcntl.h> /* definierar bland annat open() och O_RDONLY */
#include <limits.h> /* definierar bland annat SSIZE_MAX */
#include <stdio.h> /* definierar bland annat stderr */
#include <stdlib.h> /* definierar bland annat exit() */
#include <unistd.h> /* definierar bland annat read() och write() */

#define BUFFERSIZE (1024)

int main( int argc, char * argv[] )
{
  char buffer[BUFFERSIZE];
  char * oldfilename;
  char * newfilename;
  int oldfiledesc, newfiledesc;
  int read_chars, written_chars;
  int return_value; /* returvärde från close */

 /* Programmet ska ha två kommandoradsargument förutom namnet
  * vilket totalt blir 3 */
  if( 3 != argc )
  {
    printf( "Kommando: %s gammalfil nyfil\n", argv[0] );
    exit( 1 );
  }

  /* Kommer vi hit så har programmet anropats med två argument */

  oldfilename = argv[ 1 ]; /* Första argumentet är gammalfil */
  newfilename = argv[ 2 ]; /* Andra argumentet är nyfil */

  /* Försök öppna gammalfil för läsning */
  oldfiledesc = open( oldfilename, O_RDONLY );
  if( -1 == oldfiledesc )
  { fprintf( stderr, "Cannot open " ); perror( oldfilename ); exit( 1 ); }

  /* Försök öppna nyfil för skrivning */
  newfiledesc = open( newfilename,
                      O_CREAT | O_WRONLY | O_TRUNC,
                      S_IREAD | S_IWRITE );
  if( -1 == newfiledesc )
  { fprintf( stderr, "Cannot create " ); perror( newfilename ); exit( 1 ); }

  /* slinga som utför kopieringen */
  read_chars = -1;
  while( 0 != read_chars )
  {
    read_chars = read( oldfiledesc, buffer, sizeof( buffer ) );

    if( read_chars < 0 )
    { fprintf( stderr, "Cannot read " ); perror( oldfilename ); exit( 1 ); }

    if( read_chars > 0 )
    {
      written_chars = write( newfiledesc, buffer, read_chars );

      if( written_chars != read_chars )
      {
        fprintf( stderr, "Problems writing " );
        perror( newfilename );
        exit( 1 );
      }

    }
  }
  return_value = close( oldfiledesc );
  if( -1 == return_value )
  { fprintf( stderr, "Cannot close " ); perror( oldfilename ); exit( 1 ); }

  return_value = close( newfiledesc );
  if( -1 == return_value )
  { fprintf( stderr, "Cannot close " ); perror( newfilename ); exit( 1 ); }

  exit( 0 );
}
```

Figur 6. Ett filkopieringsprogram

5.2 Kommunikation via pipe

För att två processer ska kunna kommunicera via en pipe måste båda processerna ha fildeskriptorer för pipen i sina fildeskriptortabeller.

Därför skapar en huvud-process först själva pipen med systemanropet `pipe`; sedan skapar huvud-processen nya processer med `fork`. Vid `fork` ärver den nya processen fildeskriptortabellen och alla öppna filer, inklusive pipen.

Efter `fork` måste båda processerna stänga de fildeskriptorer som inte ska användas. Den process som ska läsa från en pipe, ska stänga skriv-änden av pipen. På samma sätt ska den process som ska skriva till en pipe se till att stänga läs-änden av pipen.

I Figur 10. visas två processer, en producent och en konsument, som använder en pipe för att överföra data från producenten till konsumenten.

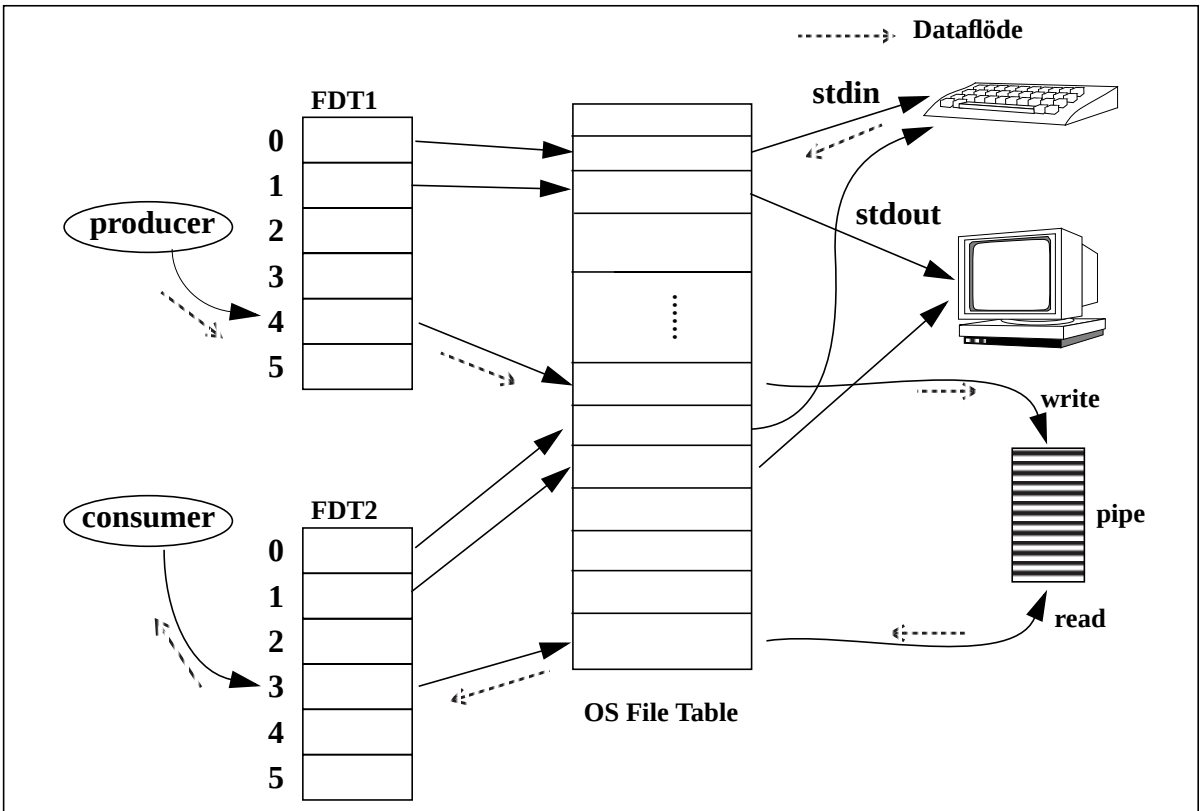
I Figur 8. och 9 finns den programkod som motsvarar Figur 10.. Producenten genererar slumptal inom intervallet `MINNUM`—`MAXNUM`. Konsumenten delar upp varje tal i primfaktorer.

```
/* pipetest.c - användning av pipe */
#include <sys/types.h> /* definierar typen pid_t */
#include <errno.h> /* definierar bland annat perror() */
#include <stdio.h> /* definierar bland annat stderr */
#include <stdlib.h> /* definierar bland annat rand() och RAND_MAX */
#include <unistd.h> /* definierar bland annat pipe() */

#define TRUE ( 1 ) /* den logiska konstanten TRUE */

#define MINNUM (   999 ) /* minsta möjliga tal som genereras */
#define MAXNUM ( 99999 ) /* största möjliga tal som genereras */

pid_t child_pid; /* för child-processens PID vid fork() */

int main()
{
  int return_value; /* för returvärden från systemanrop */
  int pipe_filedesc[2]; /* för fildeskriptorer från pipe(2) */

  fprintf( stderr, "Parent (producer, pid %ld) started\n",
           (long int) getpid() );

  return_value = pipe( pipe_filedesc ); /* skapa en pipe */
  if( -1 == return_value ) /* avsluta programmet om pipe() misslyckas */
  { perror( "Cannot create pipe" ); exit( 1 ); }

  child_pid = fork(); /* skapa en child-process */
  if( 0 == child_pid )
  {
    /* denna kod körs endast i child-processen (consumer) */

    int tal; /* för tal som läses från parent (producer) */

    fprintf( stderr, "Child (consumer, pid %ld) started\n",
             (long int) getpid() );

    /* stäng skrivsidan av pipen -- consumer ska bara läsa */
    return_value = close( pipe_filedesc[ 1 ] );
    if( -1 == return_value )
    { perror( "Cannot close pipe" ); exit( 1 ); }

    while( 1 )
    {
      return_value = read( pipe_filedesc[ 0 ], &tal, sizeof( tal ) );
      if( -1 == return_value ) /* read() misslyckades */
      { perror( "Cannot read from pipe" ); exit( 1 ); }

      if( 0 == return_value ) /* end of file */
      { fprintf( stderr, "End of pipe, child exiting\n" ); exit( 0 ); }

      if( sizeof( tal ) == return_value )
      {
        /* dela upp talet i primfaktorer och skriv ut dem */
        int f = 2, p = 0, inc[ 11 ] = { 1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6 };
        printf( "Consumer mottog talet %d, primfaktorer ", tal );
        while( f * f <= tal )
          if( 0 != tal % f ) { f = f + inc[ p++ ]; if( p > 10 ) p = 3; }
          else { printf( "%d ", f ); tal = tal / f; };
        printf( "%d\n", tal );
      }
      else
      {
        /* Unexpected byte count -- inform user and continue */
        fprintf( stderr, "Child wanted %d bytes but read only %d bytes\n",
                 sizeof( tal ), return_value );
      }
    }
  }
  else
  {
    /* denna kod körs endast i parent-processen */

    if( -1 == child_pid ) /* fork() misslyckades */
    { perror( "fork() failed" ); exit( 1 ); }
```

Figur 8. Programkod för processer som kommunicerar via en pipe, del 1

```
      /*
        Kommer vi hit i koden så är vi i parent-processen
        och fork() har fungerat bra
      */

      /* stäng läs-änden av pipen -- parent ska bara skriva */
      return_value = close( pipe_filedesc[ 0 ] );
      if( -1 == return_value )
      { perror( "Parent cannot close read end" ); exit( 1 ); }

      while( TRUE )
      {
        /* slumpa fram nya lagom stora tal, om och om igen */

        /* " + 0.0 " tvingar fram flyttalsberäkning i stället för heltal */
        double tmp = ( rand() + 0.0 ) / ( RAND_MAX + 0.0 );

        /* tmp har nu ett värde mellan 0 och 1 */
        int generated_number = MINNUM + tmp * ( MAXNUM - MINNUM + 0.0 );

        printf( "Producer skapade talet %d\n", generated_number );

        return_value = write( pipe_filedesc[ 1 ],
                              &generated_number,
                              sizeof( generated_number ) );
        if( -1 == return_value )
        { perror( "Cannot write to pipe" ); exit( 1 ); }

        sleep( 1 ); /* ta det lugnt en stund */
      }
    }
  }
```

Figur 9. Programkod för processer som kommunicerar via en pipe, del 2



Figur 10. Två processer som kommunicerar via en pipe

5.3 Omdirigering av stdin och stdout

Många program är skrivna så att de läser från `stdin` (till exempel med `scanf(3)`) och/eller skriver till `stdout` (med `printf(3)`). Kan man fortfarande använda dessa program för att läsa från en fil på disken eller koppla ihop dem med ett annat programs`stdin` eller `stdout`?

Du kanske har använt pipe-funktionen på kommandoraden med ”|”. Här ska vi visa hur det egentligen fungerar bakom kulisserna.

Vad man är ute efter är någon mekanism för att dynamiskt kunna omdirigera en fildeskriptor till en annan post i File Table. Detta görs med två systemanrop: dels `close` som vi redan känner till, och dels `dup2`. `close` och `dup2` används ofta i samband med `pipe`.

Manualtext för `dup2` kan finnas i sektion 2 eller i sektion 3C, oftast tillsammans med manualtext för `dup`.

5.4 Fildeskriptorduplicering

Antag att man har en stor filkatalog och att man vill veta hur många filer som finns i den. I Unix finns det ett program `ls` som skriver en lista av alla filer i en katalog till `stdout`. Ett annat program, `wc`, läser från `stdin` en text och räknar bland annat antalet rader och skriver ut det. I Unix kan man koppla ihop dessa två program med en pipe.

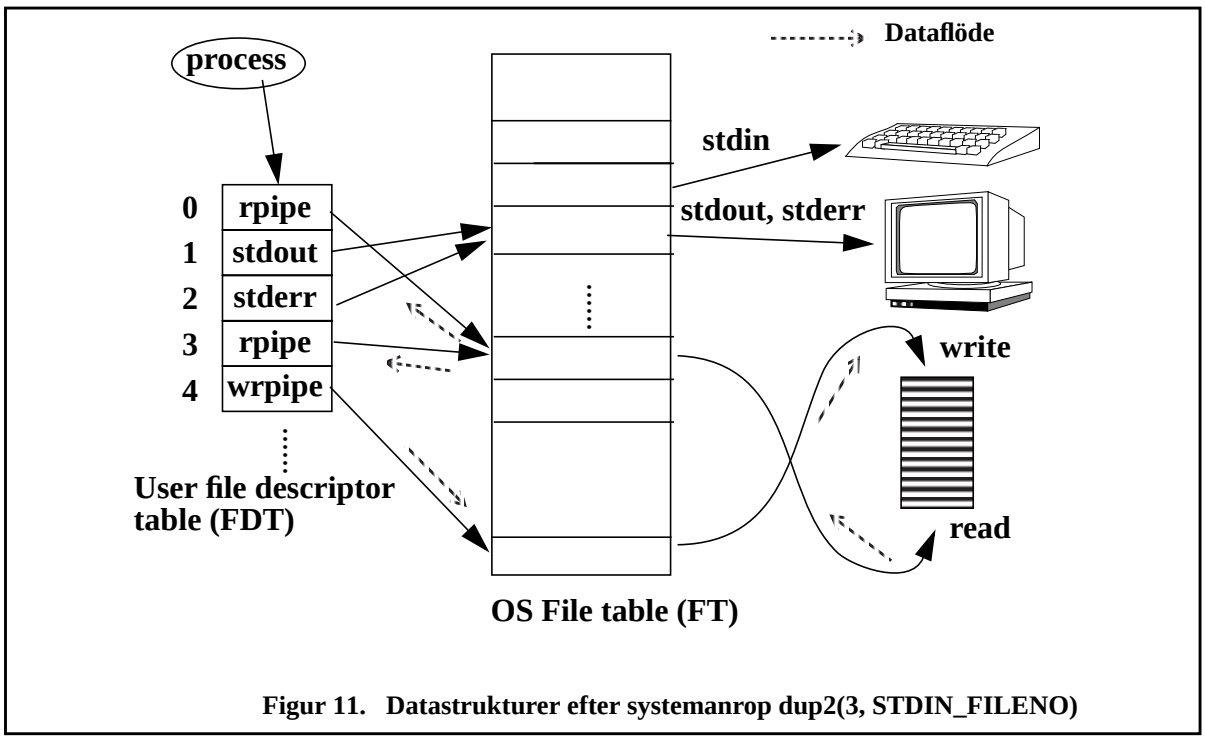
Kommanotolken (shell) översätter kommandot "`ls|wc -l`" som en begäran att sätta upp en pipe och omdirigera ls-stdout till pipens ingång och wc-stdin till pipens utgång.

5.5 Systemanropet dup2()

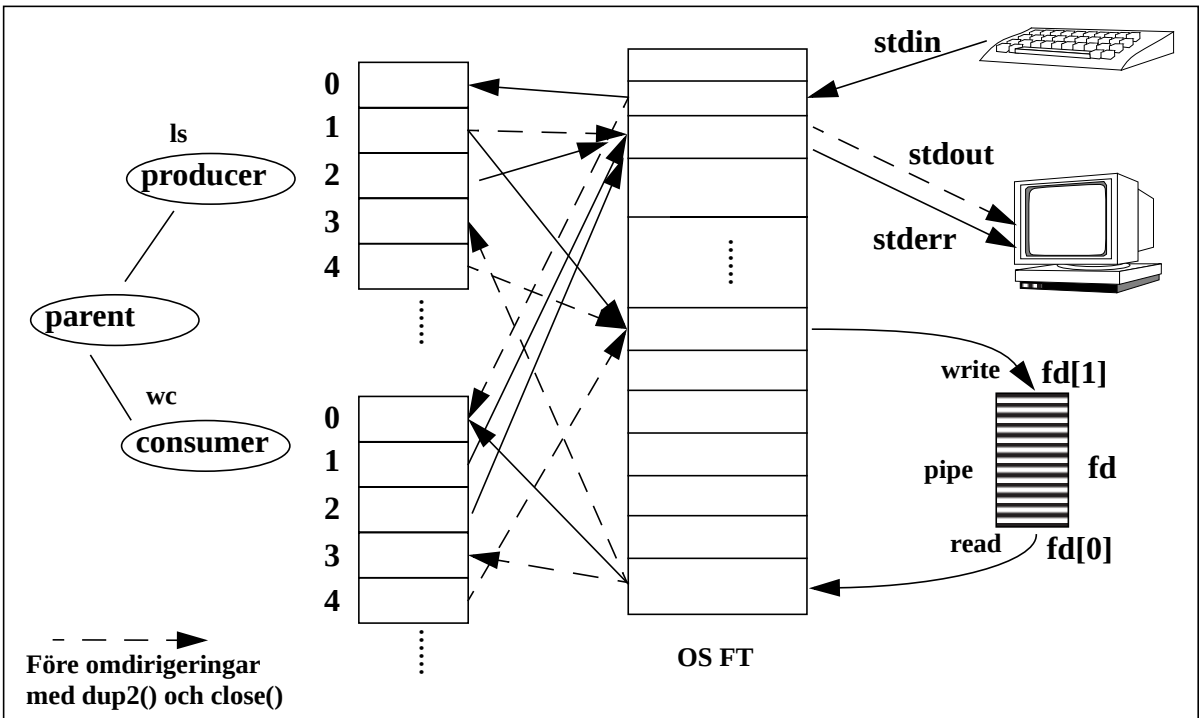
Systemanropet `dup2(oldfd, newfd)` duplicerar en befintlig fildeskriptor `oldfd` så att en process kan accessa en och samma fil (eller pipe) via två olika fildeskriptorer.

`dup2` stänger den fil som var associerad med `newfd` före anropet och kopierar därefter `oldfd` till `newfd` så att båda pekar ut samma fil och samma plats i File Table.

I Figur 11. visas exempel på en datastruktur i File Descriptor Table efter ett systemanrop till `dup2`. Makronamnet `STDIN_FILENO` är fildeskriptornumret för `stdin`, det vill säga 0.



Figur 11. Datastrukturer efter systemanrop dup2(3, STDIN_FILENO)



Figur 12. Omdirigering av stdin/stdout

Man ser att processen kommer läsa från pipens läs-ände i stället för från `stdin` (tangentbordet). Om man i sitt program efter detta gör en `scanf` så läser `scanf` från pipen i stället för från tangentbordet. Läsningen har dirigerats om.

Notera att fildeskriptor 3 inte längre används. Programmet läser från fildeskriptor 0 (som från början var `stdin`). Därför borde programmet stänga fildeskriptor 3 efter returen från `dup2`.

Notera också att figuren inte visar någon annan process som skriver till pipen. Ett program som skriver till samma pipe som det läser från är i bästa fall meningslöst. I värsta fall fastnar programmet när pipens fifo-buffert blir full. Om programmet väntar på att någon annan process ska läsa från pipen så att det blir plats i bufferten, så väntar ju programmet på något som aldrig kommer att hända.

5.6 Programexempel som visar omdirigeringar

I Figur 12. visas en mer komplicerad användning av pipes för kommunikation.

En parent-process har skapat två child-processer som kommunicerar via en pipe. Pipens ingång och utgång har dirigerats om till child-processernas stdout respektive stdin.

I Figur 13. och 14 visas programmet som åstadkommer detta med systemanrop. Programmet fungerar så här:

Skapa en pipe. Pipens läs-ände är `pipe_filedesc[ 0 ]`; skriv-änden är `pipe_filedesc[ 1 ]`.

Gör två systemanrop fork.

Den process som blir "ls" gör dup2 och kopierar `pipe_filedesc[ 1 ]` till stdout (fildeskriptor nr 1). På så sätt blir stdout kopplad till pipens ingång `pipe_filedesc[ 1 ]`.
Sedan stängs oanvända fildeskriptorer med `close`. Till sist gör processen `execl`. All utmatning från `ls` kommer att skickas via fildeskriptor nummer 1 och alltså komma till pipens ingång.

```
/* duptest.c - användning av dup2() för omdirigering */

#include <sys/types.h> /* definierar typen pid_t */
#include <sys/wait.h> /* definierar bland annat WIFEXITED */
#include <errno.h> /* definierar errno */
#include <stdio.h> /* definierar bland annat stderr */
#include <stdlib.h> /* definierar bland annat rand() och RAND_MAX */
#include <unistd.h> /* definierar bland annat pipe() och STDIN_FILENO */

#define PIPE_READ_SIDE ( 0 )
#define PIPE_WRITE_SIDE ( 1 )

pid_t childpid; /* för child-processens PID vid fork() */

int main()
{
  int pipe_filedesc[ 2 ]; /* för fildeskriptorer från pipe(2) */
  int return_value; /* för returvärden från systemanrop */
  int status; /* för returvärden från child-processer */

  fprintf( stderr, "Starting command: ls | wc\n" );
  return_value = pipe( pipe_filedesc ); /* skapa en pipe */
  if( -1 == return_value ) /* om pipe() misslyckades */
  { perror( "Cannot create pipe" ); exit( 1 ); }

  childpid = fork(); /* skapa första child-processen (för ls) */
  if( 0 == childpid )
  {
    /* denna kod körs endast i child-processen (för ls) */

    /* ersätt stdout med duplicerad skriv-ände på pipen */
    return_value = dup2( pipe_filedesc[ PIPE_WRITE_SIDE ],
                         STDOUT_FILENO ); /* STDOUT_FILENO == 1 */
    if( -1 == return_value )
    { perror( "Cannot dup" ); exit( 1 ); }

    /* ls ska inte läsa från pipen -- stäng läs-änden */
    return_value = close( pipe_filedesc[ PIPE_READ_SIDE ] );
    if( -1 == return_value )
    { perror( "Cannot close read end" ); exit( 1 ); }

    /* ls ska skriva till pipen. Men bara med fildeskriptor 1,
       det vill säga den som stdout hade före anropet till dup2().
       Alltså ska vi också stänga skriv-änden på pipen.
       Obs, vi tappar inte kontakten med pipen eftersom vi har kvar
       en fildeskriptor till den, på stdouts gamla plats. */
    return_value = close( pipe_filedesc[ PIPE_WRITE_SIDE ] );
    if( -1 == return_value )
    { perror( "Cannot close write end" ); exit( 1 ); }

    (void) execlp( "ls", "ls", (char *) 0 );
    /* exec returnerar bara om något fel har uppstått
       och om exec returnerar så är returvärdet alltid -1 */
    perror( "Cannot exec ls" ); exit( 1 );

    /* slut på kod för första child-processen (för ls) */
  }

  /* Denna kod körs endast i parent-processen. Det behövs inget "else"
     eftersom child-processen aldrig kan nå slutet på if-satsen.
     Fungerar child-processen bra så byter den ut detta program mot ls.
     Fungerar child-processen dåligt så avslutas den med exit( 1 ). */

  if( -1 == childpid ) /* fork() misslyckades */
  { perror( "Cannot fork()" ); exit( 1 ); }

  /* Kommer vi hit i koden så är vi i parent-processen
     och fork() har fungerat bra. */

  childpid = fork(); /* skapa andra child-processen (för wc) */
  if( 0 == childpid )
  {
    /* denna kod körs endast i child-processen (för wc) */

    /* ersätt stdin med duplicerad läs-ände på pipen */
    return_value = dup2( pipe_filedesc[ PIPE_READ_SIDE ],
                         STDIN_FILENO ); /* STDIN_FILENO == 0 */
    if( -1 == return_value )
    { perror( "Cannot dup" ); exit( 1 ); }

    /* wc ska inte skriva till pipen -- stäng skriv-änden */
    return_value = close( pipe_filedesc[ PIPE_WRITE_SIDE ] );
    if( -1 == return_value )
    { perror( "Cannot close read end" ); exit( 1 ); }
```

Figur 13. Program med omdirigering, del 1

```
      /* wc ska läsa från pipen. Men bara med fildeskriptor 0,
         det vill säga den som stdin hade före anropet till dup2().
         Alltså ska vi också stänga läs-änden på pipen.
         Obs, vi tappar inte kontakten med pipen eftersom vi har kvar
         en fildeskriptor till den, på stdins gamla plats. */
      return_value = close( pipe_filedesc[ PIPE_READ_SIDE ] );
      if( -1 == return_value )
      { perror( "Cannot close write end" ); exit( 1 ); }

      (void) execlp( "wc", "wc", (char *) 0 );
      /* exec returnerar bara om något fel har uppstått
         och om exec returnerar så är returvärdet alltid -1 */
      perror( "Cannot exec wc" ); exit( 1 );

      /* slut på kod för andra child-processen (för wc) */
    }

    /* Denna kod körs endast i parent-processen */

    if( -1 == childpid ) /* fork() misslyckades */
    { perror( "Cannot fork()" ); exit( 1 ); }

    /* Kommer vi hit i koden så är vi i parent-processen
       och fork() har återigen fungerat bra. */

    /* Parent-processen ska inte använda pipen -- stäng båda ändar */

    return_value = close( pipe_filedesc[ PIPE_READ_SIDE ] );
    if( -1 == return_value )
    { perror( "Cannot close read end" ); exit( 1 ); }

    return_value = close( pipe_filedesc[ PIPE_WRITE_SIDE ] );
    if( -1 == return_value )
    { perror( "Cannot close write end" ); exit( 1 ); }

    childpid = wait( &status ); /* Vänta på ena child-processen */
    if( -1 == childpid )
    { perror( "wait() failed unexpectedly" ); exit( 1 ); }

    if( WIFEXITED( status ) ) /* child-processen har kört klart */
    {
      int child_status = WEXITSTATUS( status );
      if( 0 != child_status ) /* child had problems */
      {
        fprintf( stderr, "Child (pid %ld) failed with exit code %d\n",
                 (long int) childpid, child_status );
      }
    }
    else
    {
      if( WIFSIGNALED( status ) ) /* child-processen avbröts av signal */
      {
        int child_signal = WTERMSIG( status );
        fprintf( stderr, "Child (pid %ld) was terminated by signal no. %d\n",
                 (long int) childpid, child_signal );
      }
    }

    childpid = wait( &status ); /* Vänta på andra child-processen */
    if( -1 == childpid )
    { perror( "wait() failed unexpectedly" ); exit( 1 ); }

    if( WIFEXITED( status ) ) /* child-processen har kört klart */
    {
      int child_status = WEXITSTATUS( status );
      if( 0 != child_status ) /* child had problems */
      {
        fprintf( stderr, "Child (pid %ld) failed with exit code %d\n",
                 (long int) childpid, child_status );
      }
    }
    else
    {
      if( WIFSIGNALED( status ) ) /* child-processen avbröts av signal */
      {
        int child_signal = WTERMSIG( status );
        fprintf( stderr, "Child (pid %ld) was terminated by signal no. %d\n",
                 (long int) childpid, child_signal );
      }
    }

    exit( 0 ); /* Avsluta parent-processen på normalt sätt */

  }
```

Figur 14. Program för omdirigering, del 2

1. Den process som blir `wc` gör `dup2` och kopierar `pipe_filedesc[ 0 ]` till stdin (det vill säga fildeskriptor nr 0). På så sätt blir stdin kopplad till pipens utgång `pipe_filedesc[ 0 ]`.
   Sedan stängs oanvända fildeskriptorer med `close` och till sist gör processen `execl`. All inmatning till `wc` kommer att läsas via fildeskriptor nummer 0, det vill säga från pipens utgång.
2. Som vanligt kontrolleras returvärdet efter varje systemanrop.
3. Ett program som skapar processer har även ansvar för att ta emot processernas returvärden från systemanropet `exit`. Parent-processen inväntar returvärden med systemanrop `wait`. Varje gång `wait` returnerar sätts parametern status med returvärdet från någon av child-processerna. Parent-processen måste anropa `wait` lika många gånger som den anropar `fork`.
4. När stdout omdirigerats till pipen så kan felmeddelanden ändå skrivas till skärmen via `stderr`, fildeskriptor nummer 2.

6.0 Litteratur

[1] A. S. Tanenbaum. *Modern Operating Systems, Second Edition*. Prentice-Hall 2001. Kapitel 10.

[2] B.W. Kerninghan, R. Pike. *The UNIX Programming Environment*. Prentice-Hall 1978, sidorna 201-225.

[3] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall 1986, sidorna 371-372.

[4] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall 1990, kapitel 2-3.