Örebro universitet
Institutionen för naturvetenskap och teknik
Försättsblad av JPZ

DATORKOMMUNIKATION OCH NÄT

Laboration 3

UNIX

Laborationens förhållande till teorin

Laborationerna **UNIX** och **Mer om UNIX och IPC** har till syfte att bl.a. lära ut hur kompilering görs i operativsystemen UNIX och Linux. Många av kommunikationsprotokollen kommer från UNIX-världen. Därför finns det en nära relation mellan datorkommunikation och UNIX som på senare tid har fått en efterföljare som kallas Linux. För nybörjaren i UNIX/Linux ger dessa laborationer också en introduktion till terminaler och kommandon.

I laborationen **UNIX** visas ett sätt för en process att starta upp andra program och att processer kan kommunicera med varandra via pipes. Tekniken kan användas för kommunikation mellan protokoll som är realiserade i form av program. Protokollen inordnas i protokollstackar. Sådana är indelade i skikt. Protokollen beskriver i detalj hur tjänsterna i skikten ska utföras. Protokollstackar beskrivs av referensmodeller för t.ex. OSI, Internet, ATM och Bluetooth. Pipes kan alltså användas för kommunikation mellan protokoll inom en och samma protokollstack.

Redovisning

Laborationen **UNIX** redovisas med en skriftlig rapport. I denna rapport ska du beskriva laborationen, redovisa dina svar och programmen som du har skrivit. Använd kursens rapportmall. Rapporten ska bestå av följande delar:

1. Titelsida
   Fyll i rubrik (laborationens titel) och personuppgifter. Gör en kort sammanfattning av laborationen.
2. Innehållsförteckning
   (Höger-klicka på innehållsförteckningen för att uppdatera.)
3. Bakgrund
   Laborationens koppling till datorkommunikation
4. Resultat
   Uppgift 2a: beskrivning av uppgiften och hänvisning till bilaga 1
   Uppgift 2b: beskrivning av uppgiften och hänvisning till bilaga 2
   Uppgift 3: beskrivning av uppgiften och hänvisning till bilaga 3
   Uppgift 4: beskrivning av uppgiften och hänvisning till bilaga 4
5. Bilagor
   1. Program enligt uppgift 2a
   2. Program enligt uppgift 2b
   3. Program enligt uppgift 3
   4. Program enligt uppgift 4

DATAKOMMUNIKATION

Laboration 3

UNIX

Kompilator

Välj mellan **CC** eller **GCC**.

Under denna laboration ska CC- eller GCC-kommandon skrivas direkt i **Terminal**. De två är likvärdiga. Det är vanligt förekommande i Linux att CC är en läkning av GCC. Val av textredigerare är oväsentligt, men en rekommendation är att använda en som kan visa radnummer för att underlätta felrättning. Dessutom är färgmärkning av kod att föredra. Därför rekommenderas **SciTE**, **utan att kompilera och länka från redigeraren**. Makefile ska **inte** användas under denna laboration.

Exempel på kompilering och exekvering

$ cc test.c
eller
$ gcc test.c
ger den körbara **a.out**. (**$** står för prompten.)

$ ./a.out
ger körning av **a.out**.

$ cc test.c –o test
eller
$ gcc test.c –o test
ger den körbara **test**.

$ ./test
ger körning av **a.out**.

Arkiv

1. Gå till **Labbar**.
2. Dekomprimera (höger-klicka > Extract Here) arkivet **UNIX-Lab.tar.gz**.
3. Döp om **UNIX-Lab** till **unix**.
4. Kopiera **unix** till **datorkom**.

Uppgifter

1. Programmet **unixfork.c** skapar en ny process (child process) som visar klockan i övre högra hörnet, samtidigt som moderprocessen (parent process) läser in och skriver ut strängar på skärmen.

   Systemanropet **int fork()** medför att en dotterprocess skapas. Denna börjar exekvera samma kod som moderprocessen, från instruktionen efter fork, samtidigt som moderprocessen går vidare. Fork returnerar olika pid (process-id, processnummer) i moder- och dotterprocessen. I moder-proceesen returnerar fork dotterprocessens pid. Det kan exempelvis användas till att döda dotter-processen (kill). I dotterprocessen returneras alltid pid 0.

   Kompilera och provkör! Se handledningen till Linux för kompilering och körning.

2. a) Bryt ut funktionen show_time ur unixfork.c och låt den bilda ett eget program. Kompilera och kör det i **bakgrunden**. Med kommandot **ps a** (process status) kan du se vilken pid det fick och med kommandot **kill** kan du stoppa det. Se handledningen till Linux för parametrar/switchar till **kill**. Testa även detta. (Program som **inte** körs i bakgrunden kan avbrytas med Ctrl/C.)

   b) Systemanropet **exec** laddar in och kör ett program. Det finns ett antal varianter av exec, exempelvis

   ```
   int execlp(char *file, *arg0, [*arg1, ..., *argn] *argn+1)
   ```

   där file och arg0 ska vara adressen till en teckensträng som innehåller programnamnet och sista parametern måste vara NULL. Parametrarna arg0 - argn skickas med till det nya programmet. För att exempelvis köra igång programmet prog2a, som inte har några parametrar skriver man alltså:

   ```
   execlp("./prog2a", "prog2a", NULL);
   ```

   Observera att execlp behöver headerfilen **unistd.h**. Inkludera denna i källkoden med

   ```
   #include <unistd.h>
   ```

   Efter fork är det vanligt att man låter dotterprocessen dra igång ett nytt program.

   Ändra unixfork.c så att att du startar det nya programmet (2 a) med hjälp av execlp, istället för att anropa funktionen show_time.

3. Unix-processer kan kommunicera med hjälp av rör (pipes).

   Funktionen **pipe** skapar ett rör och returnerar en vektor med två fildeskriptorer, fd[0] för läsning och fd[1] för skrivning.

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

   Funktionsvärdet blir 0 om allt gick bra, -1 annars. Pipe måste anropas före fork, så att båda processerna känner till fildeskriptorerna. Kommunikationen sker därefter med systemanropen read och write.

   Read och write returnerar antalet bytes som lästs resp skrivits. Vid fel returneras -1.

   ```
   int read(int fd, char *buf, int nbyte);
   int write(int fd, char *buf, int nbyte);
   ```

   Read och write kan användas på både filer, rör och sockets.

   Anm: Systemanropet **int open(char *path, int flags, int mode)** öppnar/skapar en fil för läsning/skrivning och systemanropet **int socket(int domain, int type, int protocol)** skapar en ändpunkt för en kommunikationslänk. Båda returnerar en fildeskriptor.

   Fildeskriptorn **fd** anger var läsning resp skrivning sker. Vid läsning från ett rör ska alltså fd[0] användas, och vid skrivning till ett rör används fd[1]. **Buf** är adressen till ett teckenfält där det som ska skrivas ut finns resp det som läses in hamnar. **Nbytes** anger hur många bytes som ska skrivas resp läsas.

   På följande sida visas ett exempel med read, write och fildeskriptorer. Observera att funktionen strlen behöver headerfilen **string.h**. Några av headerfilerna i exemplet är överflödiga men kan behövas i fullständiga lösningar.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <time.h>
#include "vt200.h"

main()
{
   int fd[2], pid, n;
   char buf[100], str[100];
   ...
   pipe(fd);
   pid = fork();
   if (pid == -1) {
      ...
   }
   else if (pid == 0) {
      ...
      n = read(fd[0], buf, sizeof(buf));
      ...
   }
   else {
      ...
      n = write(fd[1], str, strlen(str)+1);  /* Inkl. '\0' */
      ...
   }
   ...
}
```

Skriv ett C-program som skapar ett rör och en dotterprocess. Moderprocessen ska därefter läsa in textsträngar från tangentbordet och skicka över dem i röret till dotterprocessen, som skriver ut strängarna på skärmen igen. Som de flesta andra realtidsprogram ska processerna ”leva” tills användaren väljer att avbryta, i detta fall med Ctrl/D (EOF).

4. När du fått ovanstående program att fungera, inför ytterligare ett rör och en dotterprocess. Den nya processen ska läsa strängar från moderprocessen, vända dem bak-och-fram och skicka dem vidare för utskrift till den andra dotterprocessen.

KODLISTNINGAR

```
/* vt200.h */

#define ESC "\033"
#define SAVE_CURSOR printf("\0337")      /* Save current cursor position */
#define RESTORE_CURSOR printf("\0338")   /* Restore previous cursor pos. */
#define CURSOR_ON printf("\033[?25h")    /* Text cursor enabled          */
#define CURSOR_OFF printf("\033[?25l")   /* Text cursor disabled         */
#define CLRSCR printf("\033[2J")         /* Clear the entire screen      */
#define CLRLINE printf("\033[K")         /* Clear from cursor to eoln    */
#define HOME printf("\033[H")            /* Move cursor to row 1, col 1  */
#define POS(row,col) printf("\033[%0d;%0dH", row, col)
```

```
/* unixfork.c */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include "vt200.h"

void show_time()        /* This function shows actual time in                */
{                       /* upper right corner on screen.                     */
   long sec;
   struct tm *now;

   sleep(2);
   for(;;) {            /* Repeat forever!                                   */
      time(&sec);       /* Time in seconds since January 1, 1970             */
      now = localtime(&sec);    /* Time in hours, minutes, seconds, ...      */
      CURSOR_OFF;
      SAVE_CURSOR;
      POS(1,70);
      printf("%02d:%02d:%02d", now->tm_hour, now->tm_min, now->tm_sec);
      RESTORE_CURSOR;
      CURSOR_ON;
      fflush(stdout);
      sleep(1);
   }
}

void interaction()            /* This function asks for a string and         */
{                             /* prints it out again, until Ctrl/D is        */
   char str[60];              /* pressed (EOF).                              */

   POS(15,10); printf("A string please: "); CLRLINE;
   fflush(stdout);
   while(scanf("%s", str) != EOF) {
      POS(15,10); printf("The string was: %s", str); CLRLINE;
      POS(17,10); printf("New string please: "); CLRLINE;
      fflush(stdout);
   }
}

main()
{
   int pid;

   CLRSCR; HOME;
   POS(10,20); printf("FORKTEST, type Ctrl/D to exit!");
   fflush(stdout);

   switch (pid = fork()) {    /* Try to fork!                                 */
      case -1:
         perror("Forking");   /* Couldn't fork!                               */
         exit(1);
      case 0:                 /* This is child process.                       */
         show_time();
         break;
      default:                /* This is parent process.                      */
         interaction();       /* Child has process id = pid.                  */
         kill(pid, SIGTERM);  /* Kill child process.                          */
         break;
   }

   CLRSCR; HOME; CURSOR_ON;
   fflush(stdout);
}
```