Möte 12: Client/server-begreppet, demoner och sockets - (*PI*)

Flera processer ska köra som sagt, nu ska vi beskriva hur de körande processerna kan formera sig. En viss typ av processer brukar kallas *servrar* vilket innebär att de kör på en central punkt i ett datorsystem där flera andra processer kan komma åt dem och få del av deras tjänster. De processer som får hjälp av servrarna kallas *klienter*. En typisk server är webbservern som kör över nätet och som levererar webbsidor till webbläsare som då bli klienterna. En viktig funktion är att en server ska kunna betjäna flera klienter parallellt. Det betyder att en bra server behöver programmeras så att den kan hantera parallella händelser. En lösning är att låta servern i själva verket bestå av flera processer: en ny serverprocess skapas så fort en klient vill betjänas. Vi behöver ha något slags grundprogram körandes så en ypperlig lösning är att låta servern bestå av ett program som har en föräldraprocess körandes som lyssnar efter anslutningar. När en anslutning från en klient kommer in så skapar serverprogrammet en barnprocess som tar hand om det aktuella klientanropet samtidigt som föräldraprocessen fortsätter att lyssna efter nya anrop. Vi ska skapa ett sådant program som laboration 1. Men vi börjar att bygga upp det här från grunden. Vi behöver först ett alternativt sätt att kommunicera mellan processer som fungerar i båda riktningarna, en pipe är ju som bekant enkelriktad. Vi inför först ett mer månsidigt kommunikationsinstrument som heter *socket* och sedan ska vi undersöka hur man kan få ett program att ligga och köra hela tiden i bakgrunden och inte belasta ett körande system så mycket och bara bli aktivt i respons till en speciell händelse – vi ska skriva en så kallad demonprocess.

Sockets *(De delar av ALP som är aktuella för detta avsnitt är kapitel 5.5-5.5.5.)*

Med en pipe upprättar vi en enkelriktad kommunikationskanal. Vi skriver till en skrivdeskriptor och läser från en läsdeskriptor. Det innebär att om vi vill ha dubbelriktad kommunikation måste vi ha två pipes. Om vi dessutom har tvåvägskommunikation mellan två processer får vi två deskriptorer per pipe och alltså fyra deskriptorer och fyra stycken poster i de båda processernas fildeskriptortabeller (FDT). En pipe kräver dessutom någon form av släktskap mellan processerna. Det här är krångligt. En socket är också en kommunikationskanal som kan gå mellan två orelaterade processer och vi kan läsa och skriva till båda ändarna av socketen. Vi har alltså en enda fildeskriptor per process som processen både läser från och skriver till. Vi ska studera ett exempel av Beej på detta som belyser hur sockets fungerar i client/server-fallet. Vi har ett program som är server som lyssnar på en anslutning från en klient. När en klient ansluter så startar en loop som går tills någon av klienten eller servern avslutar, det som sker hela tiden är att användaren (som sitter på klientsidan) matar in en text som skickas till servern, när texten kommit till servern så skickas bara samma text tillbaka. En körning kan se ut så här:

```
$ ./server &
[1] 4316
$ Waiting for a connection...
ls -l
total 32
-rwxr-xr-x 1 johnny johnny 8491 Dec 27 18:10 client
srwxr-xr-x 1 johnny johnny    0 Dec 27 21:33 echo_socket
-rwxrwxrwx 1 johnny johnny 1137 Dec 27 18:31 echoc.c
-rwxrwxrwx 1 johnny johnny 1288 Dec 27 18:30 echos.c
-rwxr-xr-x 1 johnny johnny 8448 Dec 27 18:11 server
$ ./client echo_socket
Trying to connect...
Connected.
Connected.
```

```
> Hej
echo> Hej
> Papegoja!
echo> Papegoja!
> Sluta härmas!
echo> Sluta härmas!
> ^CWaiting for a connection...

$ ./client echo_socket
Trying to connect...
Connected.
> Connected.
Då kör vi igen
echo> Då kör vi igen
> Nej nu får det vara nog.
echo> Nej nu får det vara nog.
> ^CWaiting for a connection...
$
```

I början startar servern i bakgrunden och klienten startar straxt efter. Den socket som används blir synlig i filsystemet, det är därför vi kan se den med `ls -l` strax efter vi startat servern. Servern anger att den väntar på anslutningar genom att skriva ut "`Waiting for a connection...`", det sker två gånger under denna testkörning, vi avbryter nämligen klienten en gång och startar den på nytt. I klienten skrivs prompten ">" ut för att markera att klienten är redo att ta emot en ny inmatning från användaren och vi kan klart se att servern upprepar (ekar) det som användaren skriver ut.

Övningsuppgift: Ändra i servern så att innan texten skickas tillbaka baklänges till klienten. Använd nedanstående funktion på ett lämpligt sätt i servern för att åstadkomma detta.

```
reverse(char *str) //Vänder på strängen str, tex "ABCDE" blir "EDCBA"
{
  char tmp; int i;
  for(i=0;i<strlen(str)/2;i++)
  {tmp=str[i];str[i]=str[strlen(str)-i-1];str[strlen(str)-i-1]=tmp;}
}
```

Studera Beej's förklaring till hur server och klient fungerar, förklaringen finns på http://beej.us/guide/bgipc/output/html/multipage/unixsock.html läs här avsnitten 11, 11.1, 11.2, och 11.3. Kombinera detta med testkörningar av klienten och servern som är givna. Programmen för servern (`echos.c`) och klienten (`echoc.c`) finns på KTH-Social.

Kommentarer kring Beej's exempel

Vi kommer endast att studera sockets som är av STREAM-typ och då i början av slaget *UNIX*-domain. ALP har ett alternativt exempel som jag inte fick att fungera, det var därför som jag tog Beej's exempel. Texten på sidorna 116 och framåt i ALP är utmärkta referenser för att få en bra teknisk beskrivning av hur sockets fungerar. Speciellt är texten på sidan 119 väldigt betydelsefull eftersom den beskriver hur en server som använder en socket fungerar. Här är det särskilt viktigt att man inser att `accept()`-anropet levererar en ny deskriptor som hör ihop med ett specifikt klientanrop. Vi ska använda detta faktum för att utvidga funktionen kring servern så att den kan betjäna flera klienter samtidigt. Vi kan lätt se att den nuvarande servern inte alls kan betjäna flera

klienter samtidigt, om man startar en parallell klient i ett annat kommandofönster och ansluter parallellt med att man har en tidigare anslutning så står det klart att servern endast kan betjäna en av de anslutna klienterna.

Övningsuppgift: Använd `fork()` för att hantera en klient i en barnprocess. För att verkligen se att servern numera hanterar klienterna parallellt, byt ut `reverse()`-funktionen mot nedanstående:

```
reverse(char *str)
{
  char tmp; int i;
  for(i=0;i<10;i++){sleep(1); printf("%d ", i); fflush(stdout);}
  for(i=0;i<strlen(str)/2;i++)
  {tmp=str[i];str[i]=str[strlen(str)-i-1];str[strlen(str)-i-1]=tmp;}
}
```

Vi lägger alltså in en konstgjord fördröjning i serverns utförande av sin uppgift, servern pausar alltså och skriver ut talen 0 till 9 med en skeunds mellanrum. Anledningen är att vi ska observera en utskrift av talen 0 till 9 för *varje* klient som anropar servern, det är på det sättet som vi kan se att servern betjänar flera klienter samtidigt.

Det kommer att finnas ett problem med denna server och det är att när en klient är färdig och avslutar sig så kommer inte barnprocessen att kunna inväntas på ett tydligt sätt. Observera detta med hjälp av kommandot `ps`. Vi löser problemet med hjälp av signaler – om vi inför signalhanteraren

```
void sigchld_handler(int s)
{
  while(waitpid(-1, NULL, WNOHANG) > 0);
}
```

och installerar denna i huvudprogrammet med koden

```
struct sigaction sa;
sa.sa_handler = sigchld_handler; // reap all dead processes
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
  perror("sigaction");
  exit(1);
}
```

så kommer döda barnprocesser att inväntas automatiskt. Jämför din körning av servern med och utan denna kod och observera hur det blir en zombie per avslutad klient om man inte har denna kod med respektive observera hur klienterna inväntas i god ordning om denna kod finns med, det ska alltså inte uppstå några zombier då.

Demoner

Vi vill absolut köra ett serverprogram som en så kallad *demon*. En demon är en process som körs utan någon associerad terminal och med föräldraprocess 1. Det blir som en process som kör oberoende av vanliga användare och demoner startar ofta automatiskt eller med hjälp av ett skript. De är sedan tänkta att köra under lång tid. Vi vill nu skriva om vårt program som vi arbetat med ovan så att det blir en demon. Vi vill då placera socketen i `/tmp/` och vi behöver utföra ett antal åtgärder för att lyckas med detta. Läs igenom http://www.itp.uzh.ch/~dpotter/howto/daemonize för att få reda på de saker man gör för att skapa en demon.

Övningsuppgift: Skriv om servern ovan så att den blir en demon, vi kallar den hädanefter `revd`, förkortning för *reverser daemon*. Låt då socketen läggas i `/tmp/` och då måste förstås klientprogrammet också ta hänsyn till detta då vi kör igång det. Skapa vidare två skript, ett som heter `startrevd` och ett som heter `stoprevd` som startar respektive stoppar demonen. För att lyckas med detta måste du tänka igenom flera saker. Du måste hitta ett sätt att lagra demonens processid. Normalt lägger man in en fil under `/var/run/` som har demonens namn med extensionen `.pid`, alltså i vårt fall skapar vi en fil som heter `revd.pid`. Här skriver vi in processid för vår demon. Den filen skapas av skriptet som heter `startrevd` och skriptet som heter `stoprevd` öppnar den för att veta vilken process som ska avslutas.

Efter denna övningsuppgift kan du börja med laboration 2.