Möte 9: Relationer mellan körande processer - (*PI*)

Målet med att köra flera processer är ofta att få dem att samverka. Vi ska idag studera olika sätt att får processer att samverka. En viktig form av samverkan är kommunikation – man kan lätt skicka strängar till ett körande *C*-program via kommandoradsparametrarna, man skriver

```
$ ./a.out kalle lisa olle
```

till programmet som ser ut så här:

```
int main(int argc, char *argv[])
{
  int i;

  for(i=0;i<argc;i++)printf("%s ", argv[i]);
  printf("\n\n");

  return 0;
}
```

så skriver detta program ut parametrarna som också inkluderar programnamnet. Vi får körningen som, förvirrande nog, ser ut exakt likadant som anropet (så när som på att prompten, `$` saknas):

```
./a.out kalle lisa olle
```

Vi ska nu börja studera relationer mellan körande processer. Vi har tidigare sett så kallade pipar, alltså att man skriver till exempel `ls -l | grep Johnny` så kör vi `ls -l`-kommandot (och lista filer) och skicar resultatet till kommandot `grep Johnny` som skriver ut de rader som innehåller texten `Johnny`. Det här är ett vanligt sätt att kombinera processer i *UNIX* och vi ska se i detalj på vad som händer och även koppla ihop det med den tidigare teorin om processer.

Processer har någonting som heter *standard in* och *standard out* (och även *standard error*, men vi väntar med det). Detta är så kallade *strömmar* och så fort en process skapas så har den per default de tre strömmarna som är nämnda här. På strömmen *standard in* kommer det data in till den körande procssen och *standard out* och *standard error* är två utströmmar, men vi ska bara befattar oss med *standard in* och *standard out* än så länge. I exemplet ovan med `ls -l` som listade filer och `grep Johnny` som skrev ut de rader som innehåller texten `Johnny` så skapades två processer och de kopplades ihop via en pipe – en rörledning, som tog det som kom på *standard out* från `ls -l` och skickade det till *standard in* på processen `grep Johnny`. Strömmar har också en siffra (som vi senare ska kalla *fildeskriptor*) och *standard out* har 1 och *standard in* har 0. Med hjälp av lågnivåkommandona `read()` och `write()` kan vi skriva och läsa till *standard in* och *standard out* precis som med `printf()` och `scanf()` fast utan buffrar och formatering. Om vi nöjer oss med att skicka heltal så kan vi studera de enkla programmen `out.c` och `showp.c` (som vi kompilerar till `out` och `showp`):

`out.c`

```
int main(int argc, char* argv[])
{
  int a=atoi(argv[1]);
  write(1, &a, sizeof(int));
  sleep(1); return 0;
}
```

`showp.c`

```
int main() {
  int a; read(0,&a,sizeof(int));
  printf("%d\n", a);
  system("ps -o pid,ppid,pgid,sid,comm");
  return 0;
}
```

Programmet `out` tar en kommandoradsparameter och skickar på sin *standard out*. Det är det enda den gör, den fungerar som ett slags start för de program vi ska studera senare. Programmet `showp` fungerar som slutpunkten som skriver ut ett resultat och visar de körande processerna med systemanropet `ps`. Om vi bara kombinerar dessa båda program i en pipe så ser resultatet ut så här:

```
$ ./out 20 | ./showp
20
  PID  PPID  PGID   SID COMMAND
 4687  4263  4687  4687 bash
 5050  4687  5050  4687 out
 5051  4687  5050  4687 showp
 5052  5051  5050  4687 ps
```

Vi tolkar resultatet så här: programmet `out` fick 20 som kommandoradsparameter. Det skickades iväg på *standard out* som via pipen kom in till `showp`. Det programmet, i sin tur, läste det från sin *standard in* och skrev ut det. Det första vi ser är alltså utskrift av talet 20. Därefter körde `showp` systemanropet `ps` som beskrev hela sitauationen, vi ser vårt skal (`bash`) som tydligen har processid 4687, och de båda programmen `out` och `showp` som har processid 5050 respektive 5051. Slutligen ser vi också processid för `ps` som är 5052. Vi ser att skalet är förälder till `out` respektive `showp` medan `showp` själv är förälder till `ps`.

Vi skapar nu ett till program som heter `double` som dubblerar det den får in och skickar vidare resultatet. Vi skriver det så här:

```
int main()
{
  int a;
  read(0,&a,sizeof(int));
  a=2*a;
  write(1,&a,sizeof(int));
  sleep(1);
  return 0;
}
```

och om vi kombinerar detta med de tidigare programmen i pipen

```
$ ./out 20 | ./double | ./showp
```

så kan vi få körningen

```
40
  PID  PPID  PGID   SID COMMAND
 4687  4263  4687  4687 bash
 5066  4687  5066  4687 out
 5067  4687  5066  4687 double
 5068  4687  5066  4687 showp
 5069  5068  5066  4687 ps
```

Och detta kan tolkas som ovan, fast nu har vi ytterligare en spelare med som heter `double` som, ja dubblerar det som skrivs ut innan utskriften sker.

Det intressanta börjar när vi skriver lite mer komplicerade program som också skapar egna processer. Vi ska börja med att ta bort utskriften i `showp` och vi ska inte längre påverka talet som vi

skickar genom att dubblera osv, vi ska skicka processid:n och skapa barnprocesser och se vad som händer. Vår nya `showp` ser då ut så här:

```
int main()
{
  system("ps -o pid,ppid,pgid,sid,comm");
  return 0;
}
```

Alltså bara ett anrop av `ps`.

Vi börjar med att skriva ett program som heter `spawn` som tar in ett heltal och skapar så många barnprocesser:

```
int main(int argc, char* argv[])
{
  int i,n;
  read(0, &n, sizeof(int));
  for(i=0;i<n;i++)
    if(fork()==0){sleep(30); exit(0);}
  sleep(1);
  return 0;
}
```

Om vi nu kör `./out 3 | ./spawn | ./showp` så får vi ett resultat av typen:

```
  PID  PPID  PGID   SID COMMAND
 4687  4263  4687  4687 bash
 5169  4687  5169  4687 out
 5170  4687  5169  4687 spawn
 5171  4687  5169  4687 showp
 5172  5171  5169  4687 ps
 5173  5170  5169  4687 spawn
 5174  5170  5169  4687 spawn
 5175  5170  5169  4687 spawn
```

Vi ser här de tre barnprocesserna. Om vi nu, direkt igen kör samma anrop, alltså vi ger återigen kommandot `./out 3 | ./spawn | ./showp` så kan resultatet se ut så här:

```
  PID  PPID  PGID   SID COMMAND
 4687  4263  4687  4687 bash
 5173     1  5169  4687 spawn
 5174     1  5169  4687 spawn
 5175     1  5169  4687 spawn
 5176  4687  5176  4687 out
 5177  4687  5176  4687 spawn
 5178  4687  5176  4687 showp
 5179  5178  5176  4687 ps
 5180  5177  5176  4687 spawn
 5181  5177  5176  4687 spawn
 5182  5177  5176  4687 spawn
```

Varför det?

PI-fråga:

Vi utökar vår arsenal av små program som gör roliga saker med processer med ett program som dödar en utvald process. Vi skriver det så här:

```
int main()
{
  int pid;
  read(0, &pid, sizeof(int));
  kill( pid, 9 );
  sleep(1);
  return 0;
}
```

Programmet läser alltså först in ett tal, ett processid, och dödar den process som har det processid:t. Vi modifierar `spawn` så att den skapar ett antal processer och skickar processid:t på en av barnprocesserna vidare, det är den barnprocessen som dödas av `killp`. Den första nya versionen av `spawn` kallar vi `spawn2` och den ser ut så här:

```
int main(int argc, char* argv[])
{
  int i,n,half, killp;
  pid_t pid;
  read(0, &n, sizeof(int));
  half = n/2;
  for(i=0;i<n;i++)
  {
    pid=fork();
    if(pid==0){sleep(30); exit(0);}
    if(i==half)killp=pid;
  }
  write(1,&killp,sizeof(int));
  sleep(1);
  return 0;
}
```

När vi nu ger kommandot `./out 6 | ./spawn2 | ./killp | ./showp` så får vi resultatet

```
  PID  PPID  PGID   SID COMMAND
 4687  4263  4687  4687 bash
 5463  4687  5463  4687 out
 5464  4687  5463  4687 spawn2
 5465  4687  5463  4687 killp
 5466  4687  5463  4687 showp
 5467  5464  5463  4687 spawn2
 5468  5464  5463  4687 spawn2
 5469  5464  5463  4687 spawn2
 5470  5464  5463  4687 spawn2 <defunct>
 5471  5466  5463  4687 ps
 5472  5464  5463  4687 spawn2
 5473  5464  5463  4687 spawn2
```

Vi får alltså en zombie... hur kan vi undvika det? Är det något problem att `ps` har PID mitt i?

PI-fråga: