## Möte 6 (Föreläsning 2)

# *Del II: Systemprogrammering och Inter-Process Communication.*

Vi repeterar ett antal preciseringar:

* En **fil** är en abstraktion som döljer krångliga detaljer om maskinvaran. En fil refererar ofta till ett fysiskt lagringsutrymme på någon form av lagringsmedium (sekundärminne såsom hårddisk, CD, DVD, Magnetband, floppy etc.) Men en fil kan också ligga i datorns primärminne då den behandlas.

Förtydligande: Inom *UNIX* är ovanstående alldeles för smalt. En **fil** i *UNIX* refererar till något som man hittar i filhierarkin och det kan vara *många många olika saker*. Mycket, väldigt mycket av ett *UNIX*-system är åtkomligt via filhierarkin. *UNIX* filhierarki är, som ni kanske minns, ett träd med roten / högst upp, under den finns katalogerna `/bin/` som innehåller körbara systemprogram som `ls` etc. Katalogen `/home/` innehåller hemkatalogerna för systemets användare. Troligtvis har ni redan något hum om detta från InfoMet.
Några exempel som illustrerar att filhierarkin är stor och omfattande är:

`/proc/`: Denna katalog innehåller underkataloger som är numrerade, vi ser till exempel katalogerna 1000, 1001 och 1002, dessa kataloger ligger inte på någon hårddisk utan är genererad information direkt från kärnan som beskriver attribut och egenskaper hos processerna med nummer 1000, 1001 och 1002. På samma sätt har varje process en underkatalog i `/proc/` som beskriver dessa attribut och egenskaper.

`/dev/`: Denna katalog innehåller *device-nodes*, alltså *enhetsnoder* som möjliggör kontakt med de anslutna enheterna till systemet, här hittar vi periferienheter som nätverkskort, hårddiskspartitioner serieportar etc.

Med kommandot `man hier` får man fullständig information om var filhierarkin innehåller. Som vi ser så är begreppet "fil" i UNIX mycket vidare än begreppet "fil" i Windows. I våra laborationer kommer vi att stifta bekantskap med UNIX filbegrepp och se att vi kan få mycket information från det. Kommandot `ps` (som visar information om processer) gräver direkt i filstrukturen under `/proc/`.

På senare år har fokus mer hamnat på katalogen `/sys/` eftersom så mycket bara skyfflats in i `/proc/` så blev den till slut stor och ostrukturerad. Det kan hända att vi kommer att undersöka detta i kursens systemprogrammeringsdel.

Vi går vidare till nästa begrepp:

* En **process** är (ungefär) ett program som körs. Program finns ofta på datorns hårddisk i filer och man kan starta program genom att tex dubbelklicka på ikoner osv, men när ett program ligger på hårddisken är det inte en process. Det är först när programmet ***kör*** som man kan tala om en process. En process är alltså något som ***pågår***. En *aktivitet*. (Vidare kan ett program starta flera processer, men mer om det senare.)

Ytterligare förtydligande: En process pågår, ja, det är sant. Men det finns också många olika sorters processer, vi kommer att ge mening åt begreppen, *demon-process*, *zombie-process*, *barn-process*, *föräldra-process*, *system-process*, *användar-process* etc. Det finns en även en populär uppfattning om *UNIX* att "allt är filer", till och med processer är filer, ja, det är en sanning med modifikation, vi kan komma åt allt i ett datorsystem som som kör *UNIX* via noder i filhierarkin, alltså filerna, men det betyder inte att vi kan identifiera allt i ett *UNIX*-system som filer. Ett tydligt exempel på detta är processerna, ja, de finns alla representerade som filer under `/proc/` men jag tycker att vi mer får anse det som en *representation* av processerna, det *är* inte processerna själv. En mer precis fomrulering av "Allt är filer i *UNIX*" skulle kunna vara att "man kan komma åt och påverka det mesta i ett *UNIX*-system via filhierarkin".

Den sista termen var vad ett operativsystem var, vi avstår dock från precisering just nu (hela kursen kan ju uppfattas som en precisering av vad ett operativsystem är):

* Ett **operativsystem** (OS) är ett grundprogram som startar då datorn slås på. OS administrerar alla resurser till datorsystemet. (Resurser: mus, skärm, minne, CPU, ja allt.) Alla program som använder någon av datorsystemets resurser måste ha en pågående dialog/korrespondens med OS.

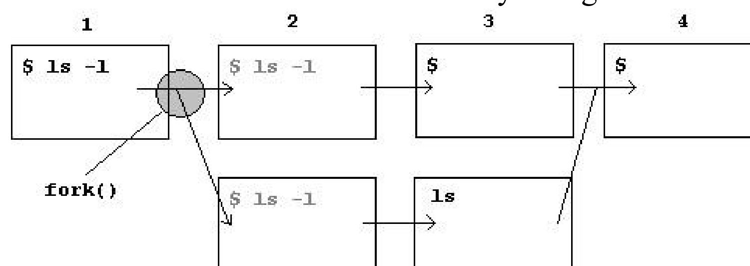
**Systemprogrammering**

Relevanta manualsidor: `fork(),wait(),exit(),getpid(),getppid(),execl(),execvp().`

Då kommer vi in på kärnan i första delen av kursen: *Systemprogrammering*. Det som skiljer systemprogrammering från grundläggande programmering (som illustrerades i *C*-kursen) är att vi måste bli mycket mer medvetna om det omgivande systemet – vi programmerar ju saker som har väldigt mycket med operativsystemet att göra.

**1. Parallella processer**

Ett datorsystem är aktivt genom sina processer, program = fil på en disk = dött, som vi sagt tidigare. process = (ungefär) körande program i datorns minne = aktivt = lever. När en ny process ska skapas i *UNIX* används alltid ett systemanrop som heter `fork()` utom då `init` skapas. Det är bara genom `fork()` som man kan skapa en ny process i *UNIX*. (Ja, utom just `init` då ja.) Eftersom allt som händer i *UNIX* händer i processer så måste också anropet ske via en process, det vill säga, för att skapa en process måste en annan process anropa `fork()`. Den som gör anropet till `fork()` kallas då förälder (*parent process*) och den process som skapas kallas barn (*child process*). Det har vi hört tidigare, men här kommer något nytt som gäller *UNIX*: *Barnprocessen är en kopia av föräldraprocessen* i det att barnprocessens processbild är exakt likadan som föräldraprocessens processbild (*process image*). För att få barnet att göra något annat måste barnets processbild bytas ut, vilket vanligtvis sker, via ett annat mycket viktigt systemanrop som heter `exec`. En vanlig situation då en ny process ska skapas är då man via ett skal kör ett kommando, till exempel skalet `bash` ska köra kommandot `ls -l`. Det kan illustreras i fyra steg:



I steg 1 ger användaren ett kommandot till skalet. Steg 2 är att skalet skapar en barnprocess med `fork()` som alltså är en kopia av skalet själv. I steg 3 byts barnprocessens processbild ut mot processbilden som hör till `ls` och sedan kör den. Skalet inväntar att barnprocessen (som förvandlats till `ls`) kört klart (en föräldraprocess behöver dock inte invänta en barnprocess). I steg 4 har barnprocessen kört klart och föräldraprocessen (skalet) är redo att ta emot ett nytt kommando. (Resutatet av kommandot "`ls -l`" är, som ni vet, en lista på filer och det resultatet har förstås presenterats på något sätt, men den presentationen är inte illustrerad ovan.)

**2. Struktur på körande processer**

Som sagt kallas en skapad process för *barn* och en skapande process kallas *förälder*. Alla processer har ett processidentifikationsnummer och alla processer har en förälder, utom en som är urföräldern och det är process nr 1 som kallas `init`. Med kommandot `ps` kan man få ett utdrag över hur det ser ut bland de körande processerna, det kan se ut så här:

```
$ ps -elfa
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root         1     0  0  75   0 -   517 -       2007 ?        00:00:10 init [2]
1 S root         2     1  0 -40   - -     0 migrat  2007 ?        00:00:02 [migration/0]
1 S root         5     1  0 -40   - -     0 migrat  2007 ?        00:00:08 [migration/1]
1 S root         6     1  0  94  19 -     0 ksofti  2007 ?        00:00:05 [ksoftirqd/1]
```

`PID` = Process IDentification. `PPID` = Parent Process IDentification. Lägg märke till att `init` är förälder till många barn. Den har själv ingen förälder utan dess PPID sätts till 0. Processen `init` är den första process som startar då systemet startar. (Läs mer om detta i DeBlanches bok.)

**3. Programmering av parallella processer i *C***

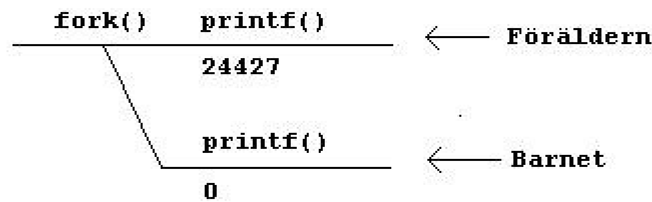
Vi ska se på hur `fork()` med mera fungerar i *C* genom att studera en rad programexempel.

| **fork1.c:** | Körning: |
|---|---|
| `#include <stdio.h>`<br>`#include <stdlib.h>`<br>`#include <sys/types.h>`<br>`#include <unistd.h>`<br>`#include <sys/wait.h>`<br><br>`int main(void)`<br>`{`<br>`  int pid;`<br>`  pid = fork();`<br>`  printf("%d\n", pid);`<br>`  return 0;`<br>`}` | <br>0<br>24427 |

*(I de kommande programexemplena kommer alla `#include`-satser att utelämnas. Namnen på programmen kommer också att förekomma som kommentarer, typ `/* fork2.c */` osv.)*

Vad gör ovanstående program? En variabel (`pid`) skapas, `fork()` anropas, returvärdet läggs i `pid` och sedan skrivs värdet på `pid` ut sedan avslutas programmet. Men hur ser körningen ut? Det är en utskrift av ***TVÅ*** olika värden? Hur kan det komma sig? Och vad betyder dessa värden? Jo, då `fork()` anropas så skapas som sagt en barnprocess. Efter anropet till `fork()` finns det alltså *två* parallella processer körandes, föräldern och barnet, båda dessa processer kör alltså `printf()`-satsen. De skiljs åt genom returvärdet från `fork()`, 0 returneras till barnet och till föräldern returneras det processid som barnet får. Det är alltså barnprocessen som skriver ut nollan och

föräldern som skriver ut 24427 som alltså är barnets PID. Vi kan illustrera det med ett tidsdiagram:




Vi tänker oss tidsaxeln som gåendes från vänster till höger. Då ser vi att `printf()` utförs av både föräldern och barnet. Här har vi även illustrerat de värden som kommer ut från de båda utskrifterna undertill. (**`24427`** respektive **`0`**.)

En viktig observation: Variabeln `pid` finns i båda processerna, men den har **olika** värden i de olika processerna! I barnet gäller `pid == 0` och i föräldern gäller `pid == 24427`! Så är det med alla variabler när man skapar ytterligare processer med `fork()`, varje variabel har ett eget förvaringsutrymme i varje process och varje barnprocess kan ha ett eget värde på de variabler som förekommer i programmet. Ett mer precist sätt att säga detta är att processerna har olika adressrymder. Adressrymderna är delar av processbilderna och eftersom barnet och föräldern är två olika processer med två olika processbilder så har vi alltså två olika adressrymder också. Det är genom returvärdet på `fork()` som vi kan skilja barn från förälder åt i *C*-koden och vi ska snart se hur det går till, men först ska vi titta på ytterligare ett exempel. Vi byter nu också ut deklarationen av processid:n till typen `pid_t`, som egentligen bara är ett annat namn för `int`. (Kan variera.)

<table>
<tr>
<td>

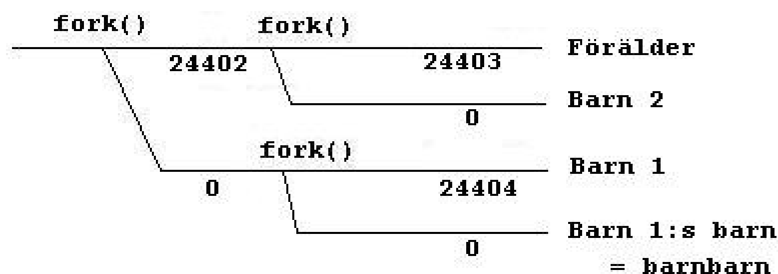
```
int main(void)  /* fork2.c */
{
  pid_t pid;

  pid = fork();
  printf("%d\n", pid);

  pid = fork();
  printf("%d\n", pid);

  return 0;
}
```

</td>
<td>

```
Körning:

0
0
24402
24403
0
24404
```

</td>
</tr>
</table>

Detta exempel liknar det föregående fast vi upprepar anropet till `fork()` och gör ytterligare en utskrift av värdet på `pid`. Man skulle då kunna tänka sig att vi får 2x2 = 4 utskrifter, men vi har *6 stycken utskrifter* som syns i programkörningen och tre är nollor, det skapas alltså **tre** barnprocesser. Hur är det möjligt? Vi gör ju bara **två** anrop till `fork()` då bör vi ju bara ha två barn... eller? Låt oss studera ett tidsdiagram för ovanstående program.



För att inse att det verkligen blir tre barn som skapas ska vi minnas att efter det första anropet till `fork()` så skapas barn 1. Efter detta första anrop till `fork()` kör både föräldern och barn 1 vidare och skriver ut 24402 respektive 0. Sedan kör *både* föräldern och barn 1 nästa anrop till `fork()` vilket resulterar i att förälder skapar barn 2 och *barn 1 skapar ett eget barn* som alltså kan tolkas som barnbarn till föräldern. Härav har vi tre barn och alltså tre nollor i utskriften.

Det sista exemplet är ganska rörigt, vi behöver finna ett sätt att programmera så att *ett kodavsnitt* i vårt program endast körs av *en process*. Vi kan åstadkomma detta på flera sätt, vi studerar först det enklaste som där vi använder att `pid` (returvärdet från `fork()`) är 0 för barnprocesserna och skilt från noll (=lika barnprocessens PID) i föräldrarna. Vi ser på exemplet nu

```
int main(void)  /* fork3.c */
{
  pid_t pid;
  pid = fork();

  if(pid == 0) { printf("Child: %d\n", pid); }
  else { printf("Parent: child pid:%d\n", pid); }

  return 0;
}
```

```
Körning:

Child: 0
Parent: child pid: 24478
```

`if`-satsen möjliggör den uppdelning i koden av "föräldrakod" och "barnkod" som vi beskrev tidigare. Nu är det lätt att se vad som sker i vilken process. Vi ser på ytterligare ett exempel:

```
int main(void)  /* fork4.c */
{
  pid_t pid;
  int tal = 1;
  pid = fork();

  if(pid == 0)
  {
    printf("Child: %d\n", pid);
    tal = 99;
    printf("Tal [child]: %d\n", tal);
  }
  else
  {
    printf("Parent: child pid:%d\n", pid);
    printf("Tal [parent]: %d\n", tal);
  }

  printf("Tal [parent/child]: %d\n", tal);
  return 0;
}
```

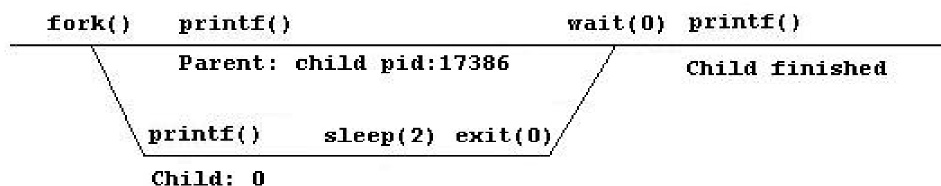
```
Körning:

Child: 0
Tal [child]: 99
Tal [parent/child]: 99
Parent: child pid:15424
Tal [parent]: 1
Tal [parent/child]:1
```

Vi har här följt uppdelningen ganska strikt, det finns egentligen bara en kodrad som både förälder och barn kör och det är den näst sista (`printf("Tal [parent/child]: %d\n", tal);`). Observera att den körs två gånger och första gången körs den innan kodrader i barnet körts klart. Så är det med parallella processer, *vi kan inte säkert veta i vilken ordning satser hörande till olika processer ska köras*. (Det är inte ens säkert att vi kan avgöra det genom att studera utskrifterna.)
Vi ska nu studera ett exempel där vi är mycket noggranna med att dela upp körningen i barnkod och föräldrakod. Vi väljer att avsluta barnet innan vi ens kommit ut ur den `if`-sats som testar på returvärdet från `fork()`:

```
int main(void) /* fork5.c */
{
  pid_t pid;
  pid = fork();

  if(pid == 0)
  {
    printf("Child: %d\n", pid);
    sleep (2);
    exit (0); //Här avslutas
              //barnprocessen helt.
  }
  else
  {
    printf("Parent: child pid:%d\n", pid);
    wait(0); //Här inväntar
             //föräldern barnets avslut
    printf("Child finished.\n\n");
  }

  return 0; //Endast föräldern gör detta anrop
}
```

```
Körning:

Child: 0
Parent: child pid:17386
<paus i 2 sekunder>
Child finished
```

Det syns inte i körningen ovan men barnet gör här en paus i 2 sekunder (`sleep(2);`) Det kan symbolisera ett stort beräkningsarbete som man väljer att lägga i en barnprocess. Det mest centrala här är två nya systemanrop: `wait()` och `exit()`. Med dessa anrop kan vi låta föräldern *invänta* barnets avslut och barnet gör avslut på ett kontrollerat sätt med `exit()`. Nytt tidsdiagram.:




Varje process som kör kan liknas vid en "tråd" som vi lägger ut och genom att använda `exit()` (i barnprocessen) och `wait()` (i föräldern) knyter vi ihop de två trådarna till en igen. Detta symboliserar att vi har så att säga delegerat en uppgift till en barnprocess, inväntat ett färdigt resultat och stängt ner barnprocessen i god ordning när uppgiften (att sova i 2 sekunder) var fullbordad. Från och med nu ska vi *alltid* göra på det här viset: en barprocess som startats ska **avslutas och inväntas** (och/eller kommuniceras med), det är endast så som en uppföljning kan ske och uppföljning bör ofta ske, föräldraprocessen har ju ett syfte med att starta en barnprocess och behöver veta om syftet är uppfyllt eller inte. (Utan `wait()` blir barnet en zombie/defunct.)

I nästa exempel ska vi införa en ännu mer formell programmeringsstil som vi hädanefter varmt rekommenderar. Alla anrop speciellt `fork()` och `wait()`, (egentligen också, `exit()`, `printf()` och `sleep()`) är så kallade *systemanrop*. Vi begär något av operativsystemet. Ibland kanske inte OS:et kan leverera det vi begär och då måste vi hantera felet. Alla systemanrop returnerar ett värde som indikerar om det har gått bra eller ej. Om `fork()` returnerar `-1` så har den misslyckats och man ska, vid korrekt systemprogrammering, (som ni ska lära er i denna kurs), ta hänsyn till det fall då ett systemanrop misslyckas. i vår fullt utbyggda formella stil ser programmet ut så här:

<table>
<tr>
<td>

```
int main(void) /* fork6.c */
{
  pid_t pid;
  pid = fork();

  switch(pid) {
    case -1:
      fprintf(stderr,"fork failed");
      exit (1);

    case 0:
      printf("I am the child with pid = %d\n",
        getpid());
      printf("My parent has pid = %d\n", getppid());
      sleep (1);
      exit(0);

    default:
      wait (0);
      printf("I am the parent, my child's pid = %d\n",
        pid);
      printf("My pid = %d.\n\n", getpid());
  }
  return 0; //Endast föräldern gör detta anrop
}
```

</td>
<td>
Körning:<br><br>I am the child with pid = 20886<br>My parent has pid = 20885<br>I am the parent, my child's pid = 20886<br>My pid = 20885.
</td>
</tr>
</table>

Här har en `switch`-sats använts för att skapa väldig klarhet kring vilken kod som hör till barn eller förälder. Vidare ser vi felhanteringen överst där programmet bryter med exitkod `1` (=parametern till `exit()`).

**4. Ordning och reda i systemprogrammeringen**

I fortsättningen kommer vi att kräva att ni kontrollerar resultatet av vissa systemanrop. Vi kommer inte att kräva att ni kontrollerar och felhanterar *alla* systemanrop (som tex `printf()`) men definitivt `fork()`. Detta är mycket viktigt av flera skäl. För det första måste ni klart kunna skilja ut vad som är ett systemanrop, vad det betyder och vad som inte är ett systemanrop. För det andra kräver felhanteringen av systemanrop att man vet hur det gick, då måste man ha testat returvärdet. Felhantering av systemanrop är en ny typ av felhantering som är mycket viktig att behärska vid systemprogrammering.

Vi ska nu se på ett avslutande exempel som visar ett av *UNIX* exekveringskommandon, `execl()`. Det finns flera kommandon av denna sort, `execv()`, `execlp()`, etc, men alla har egentligen samma underliggande `exec`-systemanrop. Se manualsidorna för en fullständig förteckning. Vi ska återknyta lite till början av denna föreläsning för att beskriva vad det är som `exec`-kommandona egentligen gör. Som tidigare nämnt, när *UNIX* skapar en ny process genom skalet så skapas den först som en kopia av det körande skalet. Därefter ersätts den skapade skalprocessen med den process som ska köras. Man brukar säga att operativsystemet "replaces current process image with a new process image" (citat från *Advanced Linux Programming*) vilket betyder ungefär att systemet byter ut den skapade skalprocessen i minnet med den nya processen som ska köras och lägger in den nya, som ska köras, i minnet istället för den som skapade den nya processen. Detta är också fallet när en annan process (än ett skal) skapar en ny process och det är precis det här "utbytandet av processbild" som `exec`-kommandona gör. Vi ser på ett program som kör `ls` i en barnprocess. Barnprocessen (som skapats med hjälp av `fork()` och som alltså är en kopia av *C*-programmet själv) byts alltså ut mot `ls`. Här är koden:

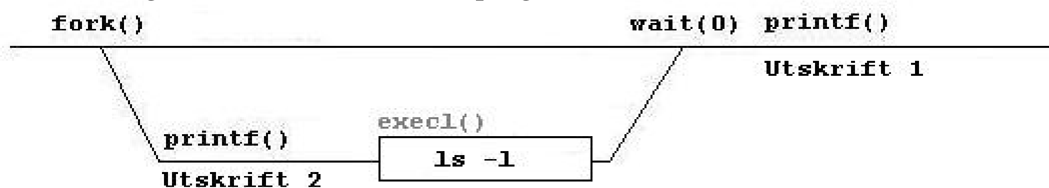
```
int main(void)  /* fork7.c */
{
  pid_t pid;
  pid = fork();

  switch(pid) {
    case -1:
      fprintf(stderr,"fork failed");
      exit (1);

    case 0:
      printf("Child with pid = %d\n", getpid());
      printf("I'm running ls:\n");
      execl("/bin/ls", "ls", NULL);
      sleep (1);   //Fråga: kommer sleep att utföras?
      exit(0);

    default:
      wait (0);
      printf("Parent, my child's pid = %d\n", pid);
      printf("My pid = %d.\n\n", getpid());
  }
  return 0; //Endast föräldern gör detta anrop
}
```

Körning:

```
Child with pid = 6410
I'm running ls:
FL4fork7.c  a.out
Parent, my child's pid = 6410
My pid = 6409.
```

Kommentar: `execl()` är ett systemanrop som alltså behöver kontrolleras... det blir din uppgift på övningen att göra detta. Ni får även undersöka detta mer noggrant på övningen.

Vi ser på ett tidsdiagram som beskriver detta program:




Vi ser lite olika utskrifter. Barnet (som är kopia av föräldern) utplånas helt från minnet och ersätts av processen `ls`. Allt som händer sedan är att `ls` kör och ger en listning av filerna. Listningen är delen ”`FL4fork7.c   a.out`” av körningen som visas till höger om källkoden ovan, två filer finns tydligen, dels ett *C*-program och dels ett körbart program (det är programmet själv respektive programmets källkod). OBS: Föräldern väntar i alla fall på att `ls` ska bli klar. Utskrift 1 kommer allra sist. (`execl()` visas i grått och `ls` i en ruta för att symbolisera att `ls` tar över platsen helt där tidigare barnprocessen fanns.) Hur är det med satserna `sleep();` och `exit(0);`? (I tidsschemat ska det vara bara `ls`, inte `ls -l`.)

## 5. Rester efter körande barnprocesser: Zombier

En process måste alltid avslutas på ett medvetet sätt. Detta är av administrationsskäl: det måste finnas en punkt då en process anses ha kört klart så att operativsystemet kan avallokera administrationsresurserna (processkontrollblock etc.) på ett solitt sätt. En process kan avsluta sig själv genom att göra `exit()`, det har vi sett ovan, eller `return 0;`. En process avslutas alltid med en så kallad "exitkod", ett värde som fångas upp som en rapport om utfallet av körningen. I korthet brukar "0" i exitkod betyda att processen lyckats och värden skilda från 0 indikerar någon

form av fel. En barnprocess som avslutats ska normalt inväntas av sin förälder, föräldern ska göra `wait()` eller motsvarande. Det blir en signal till operativsystemet att städa upp efter en avslutad process. MEN det finns tillfällen då det här inte sker. Det kan vara av flera skäl:

1. Föräldraprocessen avslutas abrupt/oväntat, kanske genom ett segmenteringsfel (pekarfel, kanske `scanf()` med glömd adressoperator (alltså `&`-tecknet)).

2. Den som programmerat föräldraprocessen har glömt att göra `wait()`.

3. Den som programmerat föräldraprocessen har skrivit ett program som visserligen gör `wait()` men på grund av något annat programmeringsfel så utförs inte `wait()`.

Icke-inväntade barn som kört klart, alltså hunnit göra `exit()` (eller `return` eller liknande) som ligger kvar i minnet kan inte längre utföra någonting. Vidare är de inte inväntade vilket betyder att OS inte rensar efter dem. De är alltså varken döda eller levande, man brukar kalla processer i detta tillstånd för *zombier* eller *defunct*-processer. Dessa tar upp utrymme och detta ska undvikas. Om ett program genererar mycket zombier måste det skrivas om och göras bättre. Vi ser på ett exempel:

**zombiecreator.c**

```
int main(void)
{
  pid_t pid;
  pid = fork();

  switch(pid)
  {
    case -1:
      fprintf(stderr,"fork failed");
      exit (1);
      break;

    case 0:
      printf("I am doing exit(), so I become a Zombie.\n");
      exit(0);

    default:
      printf("While I'm sleeping, my child [%d] is a Zombie.\n", pid);
      sleep(20);
      wait(0);
      printf("Now my child is a Zombie no more.\n");
      printf("But I'm back for some more sleep...\n");
      sleep(20);
  }
  return 0;
}
```

Vi kommer att undersöka detta program på övningen och rita ett tidsdiagram då. Vi kan bara nöja oss med att konstatera att då barnet gör `exit()` blir det en zombie under de 20 sekunder som föräldern inte inväntar det.