Möte 7: Uppföljning av föreläsningen med Peer Instruction - (*PI*)

Som sagt så kommer den här kursen endast innehålla en enda föreläsning och det var förra gången. Från och med nu så kommer vi förutsätta att ni läser innan kursmötena och att vi kan använda kursmötena till att assimilera och behandla kunskap istället för att hålla föreläsningar. Vi kommer också att presentera nytt material på ett PI-möte, men fokus kommer inte att vara där.

1. Parallella processer i *C*

Vi har ett program i *C* som vill skapa 10 barnprocesser. Vi vill kunna se alla 10 barnprocesser och föräldraprocessen med kommandot `ps -l`. Vilken principiell form på programmet är då lämplig?

A.
```
main() {
  int i; pid_t pid;

  for(i=0;i<10;i++)
    pid = fork();

  for(i=0;i<10;i++)
    if(pid==0)
      exit(0);

  for(i=0;i<10;i++)wait(0);
}
```

B.
```
main() {
  int i; pid_t pid;
  for(i=0;i<10;i++)
    {
      pid = fork();
      if(pid==0)
      {
        exit(0);
      }
    }
  for(i=0;i<10;i++)wait(0);
}
```

C.
```
main() {
  int i; pid_t pid;
  for(i=0;i<10;i++)
    {
      pid = fork();
      if(pid==0)
      {
        sleep(10);
        exit(0);
      }
    }
  for(i=0;i<10;i++)wait(0);
}
```

D.
```
main() {
  int i; pid_t pid;
  for(i=0;i<10;i++)
    {
      pid = fork();
      if(pid==0)
      {
        sleep(10);
        exit(0);
      }
    }
}
```

2. Om processbilder

Då vi kör en process så reserveras ett utrymme i datorns minne. Det finns ganska mycket detaljer kring detta och om ni vill ha mycket detaljer kan ni kolla i Rago & Stevens: *Advanced Programming in the UNIX-Environment* i 7017, men vi ska nöja oss med att kalla det som reserveras i datorns minne för en *processbild*. ALP kallar detta för *process image* och den innehåller data och kod hörande till den process som körs. Så fort en barnprocess skapas så kan vi se det som om en ny processbild skapas och eftersom barnet till en början (direkt efter anropet till `fork()`) är en kopia av föräldraprocessen (det enda som skiljer är processid) så har barnet samma uppsättning variabler som föräldern. Men alla dessa variabler (precis som allt annat som tillhör barnet) ligger i en barnets separata processbild som alltså är är skild från förälderns processbild. Givet denna information, studera följande kod:

```
main() {
  int a=10,b=20; pid_t pid;
  pid = fork();

  if(pid==0) {
    a=b;
    printf("> %d %d.\n\n", a, b);
    exit(0);
  }
  b=a;

  printf("> %d %d.\n\n", a, b);
  wait(0);
}
```

Vad blir utskriften av detta program då det kör?

A.
`> 10 10`
`> 20 20`
	B.
	`> 10 20`
	`> 20 10`

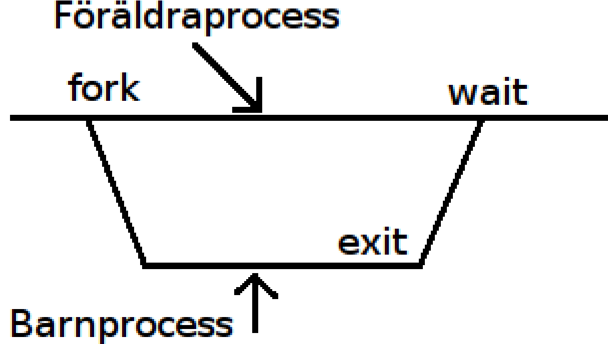
C.
`> 20 10`
`> 10 20`
	D. Inget av A-C är korrekt.

3. Skapa barn som adopteras av init

En vanlig konstruktion då man vill skapa en barnprocess är följande:

```
pid = fork();
if(pid==0)
{
  barnkod
  exit(0)
}
föräldrakod
wait(0)
mer föräldrakod som utförs när barnet är klart
```

Vi har tidigare illustrerat detta med ett diagram av följande typ:



Under barnets körning så upprätthålls alltså släktskapet mellan barn och förälder. Men vad händer om föräldern gör exit innan barnet gjort exit? Olika saker i olika fall. Vi studerar följande program:

```
main()
{
  if(fork()==0){
    printf("Child PID: %d.\n", getpid());
    sleep(100);
    exit(0);
  }

  sleep(1);
  printf("Parents exits. Do ps -a -o pid,ppid,comm.\n");
  exit(0);

}
```

Här skapar vi en barnprocess och avslutar föräldern omedelbart. En testkörning ser ut så här:

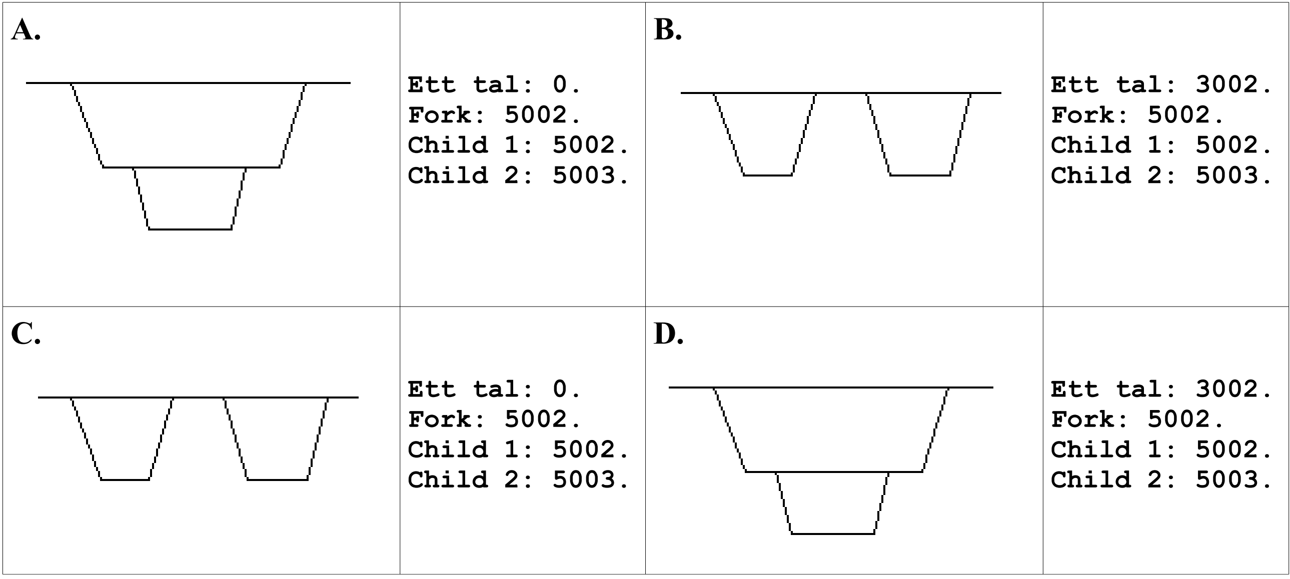
```
$ ./a.out
Child PID: 4567.
Parents exits. Do ps -a -o pid,ppid,comm.
$ ps -a -o pid,ppid,comm
  PID  PPID COMMAND
 4233  4224 emacs
 4497  4429 man
 4500  4497 sh
 4501  4500 sh
 4505  4501 less
 4567     1 a.out
 4568  4524 ps
```

Vi ser att vår `a.out`, som alltså kör `sleep(100)`, har fått en ny förälder: den med processid = 1. Processen med processid = 1 är en speciell process, det är den första processen som kör, den heter *init* och är urförälder till alla körande processer. Då en förälder avslutar sig själv utan att ha låtit sitt barn köra klart adopteras under vissa förhållanden barnet av *init*. Men inte alltid: gör så här, öppna en konsol och ge kommandot sleep 100. Öppna ytterligare en konsol och ge kommandot `ps -a -o pid,ppid,comm`. Då ser vi processer som gör sleep. Avsluta nu den konsol där kommandot sleep 100 gavs. Är barnet adopterat av *init*? Troligen inte! Hur kan vi försäkra oss om att init adopterar barn som inte kört klart då deras föräldrar avslutar?

4. Vi tittar på en gammal tentamensfråga. Betrakta nedanstående program. Vi antar i det här fallet att programmets processid blir runt 5000.

```
int main(void) {
  pid_t pid1, pid2, twothousand = 2000;

  pid1 = fork();

  //Barn 1:
  if(pid1==0) {
    childpid1 = getpid();
    printf("Child 1: %d.\n", getpid());

    pid2 = fork();

    //Barn 2:
    if(pid2==0) {
      printf("Child 2: %d.\n", getpid());
      exit(0);
    }

    wait(0);
    exit(0);

  }

  //Föräldern:
  printf("Ett tal: %d.\n", pid1 - twothousand);
  printf("Fork: %d.\n", pid1);

  wait(0);

  return 0;
}
```

Följande alternativ finns till möjliga tidsdiagram och utskrifter. Endast ett är rimligt.



A.	Ett tal: 0. Fork: 5002. Child 1: 5002. Child 2: 5003.	**B.**	Ett tal: 3002. Fork: 5002. Child 1: 5002. Child 2: 5003.
C.	Ett tal: 0. Fork: 5002. Child 1: 5002. Child 2: 5003.	**D.**	Ett tal: 3002. Fork: 5002. Child 1: 5002. Child 2: 5003.

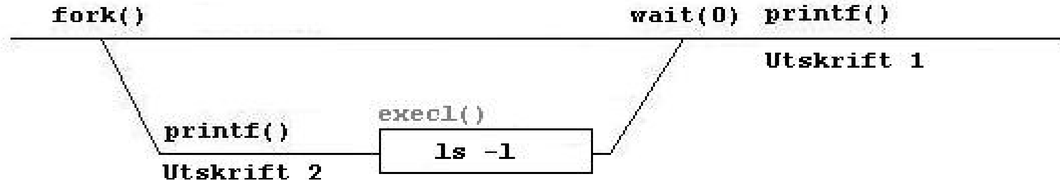
5. Vi studerar igen koden till `fork7.c`, vi ska diskutera en del kring de processer som uppkommer då det programmet kör. (Handout!)

```
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.
```

Vi ser på ett tidsdiagram som beskriver detta program:



Frågan ligger i koden: Kommer sleep att utföras? Varför, varför inte? Hur är det med exit(0);

Finns det några förhållanden liknande de ovan där sleep respektive exit kommer att utföras? Hur är det med kommandoföljden

```
exit(0);
sleep(1);
```

6. Zombier

Vi ska undersöka ett lite annorlunda zombieprogram än det vi arbetade med på övningen, vi tar in kommandoradsargument som ett tal och lägger det i variabeln `nz`.

```
int main(int argc, char *argv[])
{
  int i, nz = atoi(argv[1]); //Vi tar in kommandoradsarg 1 och lägger den i nz

  for(i=0;i<nz;i++)if(fork()==0)exit(0);

  printf("Run 1:\n===================\n"); system("ps -o stat,pid,ppid,comm");
  sleep(1);

  for(i=0;i<nz/2;i++)wait(0);

  printf("Run 2:\n===================\n"); system("ps -o stat,pid,ppid,comm");
  sleep(1);

  for(i=0;i<nz/2;i++)wait(0);

  printf("Run 3:\n===================\n"); system("ps -o stat,pid,ppid,comm");

  return 0;
}
```

Vad gäller angående detta program?

A. Ett fixt antal processer skapar alltid, hälften av dem blir zombier.
B. Olika antal processer kan skapas i olika programkörningar och alla blir zombier direkt.
C. Olika antal processer kan skapas i olika programkörningar och hälften blir zombier direkt.
D. Inget av ovanstående gäller.

Ur manualsidan till wait:

A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by init(8), which automatically performs a wait to remove the zombies.