Tentamen i Operativsystem, HI1025:LAB1 - 15 mars, 2017

Allmänna instruktioner. Tentamen innehåller 3 programmeringsproblem av den art vi exemplifierat på seminarier och i övningar. För godkänt betyg ska alla problem lösas. Alla processer som skapas måste inväntas och inga zombier (`<defunct>`) får synas i utskrifter av processers status/relationer.

Uppgifter

(1) Nedanstående program tar in två kommandoradsargument. Utvidga programmet så att det startar så många barnprocesser som anges av det första argumentet. Alla dessa barnprocesser ska läggas som barn under `init`. Alla barnen ska göra en `sleep()` av en viss längd men ett av barnen, vars ordningsnummer anges av de andra kommandoradsargumentet, ska göra en kortare `sleep()` och alltså avslutas innan alla de andra.

Systemanropet `system("ps -e -o pid,ppid,pgid,sess,comm | grep uppg1")` ska användas vid tre tillfällen för att verifiera att programmet fungerar precis som det ska: 1. Då samtliga barn skapats och ligger under `init`, 2. Då ett av barnen avslutats. 3. Då samtliga barn avslutats. Dessa anrop finns i koden och får inte ändras. Programmet ser ut så här från början, sista raderna är de som inte får ändras, de ska ge den utskrift som krävs. I övrigt får vad som helst läggas till för att få programmet att fungera enligt kraven.

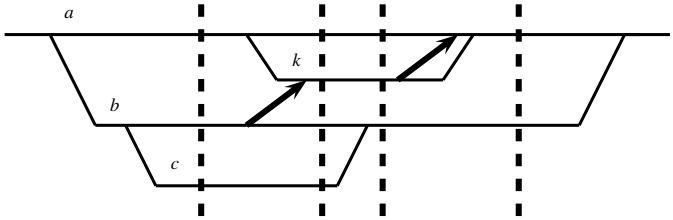
```
#include <stdlib.h>
main(int argc, char *argv[])
{
  int i, n, d; n=atoi(argv[1]); d=atoi(argv[2]);
  sleep(1);
  system("echo ============================== 1 second passed");
  system("ps -e -o pid,ppid,pgid,sess,comm | grep uppg1");
  sleep(2);
  system("echo ============================== 3 seconds passed");
  system("ps -e -o pid,ppid,pgid,sess,comm | grep uppg1");
  sleep(2);
  system("echo ============================== 5 seconds passed");
  system("ps -e -o pid,ppid,pgid,sess,comm | grep uppg1");
}
```

En godkänd testkörning: *(Inga zombier!)*

```
$ ./uppg1 4 2
============================== 1 second passed
 1942  1695  1942  1695 uppg1
 1944     1  1942  1695 uppg1
 1946     1  1942  1695 uppg1
 1948     1  1942  1695 uppg1
 1950     1  1942  1695 uppg1
============================== 3 seconds passed
 1942  1695  1942  1695 uppg1
 1944     1  1942  1695 uppg1
 1948     1  1942  1695 uppg1
 1950     1  1942  1695 uppg1
============================== 5 seconds passed
 1942  1695  1942  1695 uppg1
```

och här ser vi att programmet anropas med 4 respektive 2 som kommandoradsargument, 4 barnprocesser (1944, 1946, 1948, och 1950) skapas och läggs under `init` och den andra barnprocessen (1946) avslutas och de andra fortsätter köra lite grann och till sist är alla avslutade. Kompilera med `gcc uppg1.c -o uppg1` för att få processerna att heta just `uppg1` så att de kan fångas upp av `grep`. Filen `uppg1.c.` innehåller koden ovan, redigera den och lämna in den som svar.

(2) Skriv ett program som skapar ett antal processer vars relationer beskrivs av nedanstående tidsdiagram



Processen `k` ska defineras av följande kod i `killer.c` (som inte får ändras):

```
int pid;
while(read(0,&pid,sizeof(int))) {
  if(kill(pid,9)==0)write(1,&pid,sizeof(int));
  else {pid=-pid; write(1,&pid,sizeof(int));}
}
```

Processen `k` är alltså en "killer" som dödar den process vars processid den får på *stdin*. Hela programmet ska fungera så här: Processen `a` skapar processerna `b` och `k` som barn. Processen `b` skapar i sin tur barnet `c` och skickar `c`'s processid via en pipe till `k`. Efter `k` har försökt döda `c` så skickar `k` sitt resultat via en annan pipe till `a` som inväntar både `b` och `k`. De fyra tjocka lodräta streckade strecken utgör checkpunkter där `a` ska göra anropet `system("ps -o pid,ppid,comm");` för att kontrollera att processrelationerna är så som föreskrivs av tidsdiagrammet. Alltså, vid första system/ps-anropet ska `a`, `b` och `c` synas, vid andra anropet ska `a`, `b`, `c` och `k` synas, vid tredje ska `a`, `k` och `b` synas, vid fjärde (sista) anropet ska `a` och `b` synas. Du behöver experimentera med väl valda anrop till `sleep()` för att få det att fungera perfekt. (De tjocka heldragna pilarna representerar förstås de två pipes som behövs i programmet.) Vid ett väl valt tillfälle ska huvudprocessen `a` skriva ut processid på det barn som dödas, det är ju det som skickas från `k` till `a`. (Nedan ser vi det i och med `Child killed:  3839.`.)

Ledning: Det finns utförligt exemplifierat i exempelprogrammen som följer med hur man inkluderar externa program (som `killer.c`) inklusive omdirigeringen

En provkörning kan se ut så här: *(Inga zombier!)*

```
$ ./uppg2
  PID  PPID COMMAND
 3792  2921 bash
 3836  3792 uppg2
 3837  3836 uppg2
 3838  3836 sh
 3839  3837 uppg2
 3840  3838 ps
  PID  PPID COMMAND
 3792  2921 bash
 3836  3792 uppg2
 3837  3836 uppg2
 3839  3837 uppg2
 3841  3836 uppg2
 3842  3836 sh
 3843  3842 ps
  PID  PPID COMMAND
 3792  2921 bash
 3836  3792 uppg2
 3837  3836 uppg2
 3841  3836 k
 3844  3836 sh
 3845  3844 ps
Child killed: 3839.
  PID  PPID COMMAND
 3792  2921 bash
 3836  3792 uppg2
 3837  3836 uppg2
 3846  3836 sh
 3847  3846 ps
```

Utskrifternas fullständighet kan variera något, bland annat kanske inte processen `sh` (som uppstår i samband med anropet till `system("ps ...")`) finns med. Det viktiga är att er statusutskrift (som exemplifieras till vänster) ska överenstämma med tidsdiagrammet ovan så första anropet till `system("ps ...")` verkligen visar processerna `a`, `b` och `c` men inte `k`, andra anropet ska visa alla fyra (`a`, `b`, `c` och `k`), tredje ska bara visa `a`, `b` och `k` och sista anropet ska bara visa `a` och `b`. Det är frågan om att lägga in `sleep(...)` på väl valda ställen i de olika ingående processerna för att få det att fungera.

(3) I laboration 3 studerades *deadlock* mellan ett godtyckligt antal körande trådar. Vi ska se på en enklare variant med bara två trådar men med en alternativ strategi för att undvika deadlock. Studera nedanstående två trådprogram (de finns förstås bland materialet):

```
// uppg3a.c:

int thread=0, run=20;
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
pthread_t t1, t2, decr_thr;

int choose(int n){return rand()%n+1;}

void* tf1(void *par) {
  while(run>0) { srand(2*time(0));
    int dur; sleep(choose(3)); if(!run)break;
    pthread_mutex_lock(&m1); sleep(1);
    pthread_mutex_lock(&m2);
    dur=choose(3); while(dur--) {thread=1; sleep(1);} thread=0;
    pthread_mutex_unlock(&m1); pthread_mutex_unlock(&m2);
  }
  pthread_exit(0);
}

void* tf2(void *par) {
  while(run>0) { srand(3*time(0));
    int dur; sleep(choose(3)); if(!run)break;
    pthread_mutex_lock(&m2); sleep(1);
    pthread_mutex_lock(&m1);
    dur=choose(3); while(dur--) {thread=2; sleep(1);} thread=0;
    pthread_mutex_unlock(&m1); pthread_mutex_unlock(&m2);
  }
  pthread_exit(0);
}

void* decr(void* par){
  while(run--)sleep(1); pthread_cancel(t1); pthread_cancel(t2);
  pthread_exit(0);
}

main(){
  srand(time(0));
  pthread_create(&t1,NULL,&tf1,NULL); pthread_create(&t2,NULL,&tf2,NULL);
  pthread_create(&decr_thr,NULL,&decr,NULL);
  while(run>0){ printf("%d:%d.\n", run, thread); sleep(1);}
  pthread_join(t1,NULL); pthread_join(t2,NULL);
  pthread_join(decr_thr,NULL);
}
```

Då programmet körs skapas ofta deadlock. Det finns två mutexar och en tråd anger att den har båda genom att sätta den globala variabeln `thread` till sitt nummer, antingen 1 eller 2. Denna skrivs ut av huvudprogrammet och om en 1:a skrivs ut betyder det alltså att tråd nr 1 har båda mutexarna, om en 2:a skrivs ut betyder det att tråd nr 2 har båda mutexarna. Det finns en tredje tråd också vars enda uppgift är att räkna ner variabeln `run` och när den når 0 så avlutas båda trådarna. Så är i alla fall tanken, men detta program fungerar inte så bra, deadlock skapas nämligen ganska ofta som sagt och det uttrycker sig genom att utskriften ofta bara blir en lång rad av nollor.

Uppgiften består i att förstå och uttrycka varför detta program skapar deadlock och varför nästa program inte skapar deadlock. I uppgiften finns ett anrop till `pthread_cancel()`, det enda det systemanropet gör här är att begära att en tråd blir avslutad.

Vi jämför detta program med nästa program som är en uppgraderad variant:

```
// uppg3b.c

int thread=0, run=20;
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
pthread_t t1, t2, decr_thr;

int choose(int n){return rand()%n+1;}

void* tf1(void *par) {
  while(run>0) { srand(2*time(0));
    int dur; sleep(choose(3)); if(!run)break;
    pthread_mutex_lock(&m1);
    if(!pthread_mutex_trylock(&m2)) {
      dur=choose(3); while(dur--) {thread=1; sleep(1);} thread=0;
      pthread_mutex_unlock(&m1); pthread_mutex_unlock(&m2);
    }
    else pthread_mutex_unlock(&m1);
  }
  pthread_exit(0);
}

void* tf2(void *par) {
  while(run>0) { srand(3*time(0));
    int dur; sleep(choose(3)); if(!run)break;
    pthread_mutex_lock(&m2);
    if(!pthread_mutex_trylock(&m1)) {
      dur=choose(3); while(dur--) {thread=2; sleep(1);} thread=0;
      pthread_mutex_unlock(&m1); pthread_mutex_unlock(&m2);
    }
    else pthread_mutex_unlock(&m2);
  }
  pthread_exit(0);
}

void* decr(void* par){
  while(run--)sleep(1); pthread_cancel(t1); pthread_cancel(t2);
  pthread_exit(0);
}

main(){
  srand(time(0));
  pthread_create(&t1,NULL,&tf1,NULL); pthread_create(&t2,NULL,&tf2,NULL);
  pthread_create(&decr_thr,NULL,&decr,NULL);
  while(run>0){ printf("%d:%d.\n", run, thread); sleep(1);}
  pthread_join(t1,NULL); pthread_join(t2,NULL);
  pthread_join(decr_thr,NULL);
}
```

Båda programmen finns med i det material som kommer med själva tentan. Det finns en viktig skillnad i körningen mellan dessa båda program. På nästa sida ges ett antal provkörningar av de båda programmen och du kan själv ägna tid åt att provköra programmen. Den stora skillnaden mellan programmen är att det första programmet, som kallas `uppg3a.c` ofta slutar i att skriva ut en lång serie nollor, dvs ingen av de två arbetstrådarna har tillgång till båda mutexarna samtidigt. Medan det andra programmet, som kallas `uppg3b.c`, skriver ut både 1:or och 2:or under hela sin livstid. Därmed lyckas trådarna i `uppg3b.c` äga båda mutexarna under hela programmets körning.

Testkörning 1:

```
$ ./uppg3a
20:0.
18:0.
17:2.
16:2.
15:0.
14:0.
13:0.
12:0.
11:0.
10:0.
9:0.
8:0.
7:0.
6:0.
5:0.
4:0.
3:0.
2:0.
1:0.
$
```

Testkörning 2:

```
$ ./uppg3a
20:0.
18:0.
17:1.
16:0.
15:2.
14:2.
13:0.
13:0.
11:1.
10:1.
9:1.
9:0.
7:0.
7:0.
5:1.
4:0.
3:2.
2:2.
1:2.
$
```

Testkörning 3:

```
$ ./uppg3b
20:0.
18:0.
17:0.
16:1.
15:1.
14:0.
13:2.
12:0.
11:0.
10:0.
9:0.
8:1.
7:1.
6:1.
5:0.
4:0.
3:2.
2:2.
1:0.
$
```

Testkörning 4:

```
$ ./uppg3b
20:0.
18:0.
17:0.
16:2.
15:1.
14:1.
13:0.
12:2.
11:2.
10:2.
9:0.
8:0.
7:0.
6:1.
5:1.
4:0.
3:2.
2:2.
1:0.
$
```

I de två första kolumnerna anges testkörningar av det första programmet. Som vi ser avslutas den första testkörningen med en lång serie nollor, det betyder att ingen av trådarna äger båda mutexarna. Den andra körningen, också av det första programmet, är i själva verket en ganska ovanlig körning, här skrivs det ut både ettor och tvåor under hela programmets livslängd. Men väldigt ofta när det första programmet körs, blir det till och med *bara* nollor! Men de andra två kolumnerna, som visar testkörningar av det andra programmet visar inte bara nollor, utan i båda körningarna finns ettor och tvåor under hela detta programs livslängd, i själva verket, då det andra programmet körs, blir det *alltid* ettor och tvåor under hela programmets livslängd.

Din uppgift är att förklara varför det första programmet (`uppg3a.c`) ofta bara ger en massa nollor medan det andra programmet (`uppg3b.c`) *alltid* ger både ettor och tvåor under hela sin livslängd. Förklara detta med utgångspunkt från de båda programmens källkoder samt referenser till manualsidan för `pthread_mutex_lock/trylock/unlock`. Ge svaret i textform istället för att skriva några nya program. (Självklart kan du experimentera med att ändra i koden till programmen `uppg3a/b.c` men inget program kommer att bedömas som du lämnar in. Det som bedöms är din förklaring av programmens olika beteenden.

(Det enda ni behöver veta om anropet `pthread_cancel` är att det begär att en tråd avslutas, grubbla inte över det.)