Tentamen i Operativsystem, HI1025:LAB1 - 9 juni, 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 lösningar behöver inte fungera perfekt dock, möjlighet till kompletteringar kan ges så gör ordentliga försök på alla uppgifter. Alla processer som skapas måste inväntas och inga zombier (`<defunct>`) får synas i utskrifter av processers status/relationer.

Uppgifter

(1) I kursen har vi studerat pipes och sockets noggrannt, de är båda exempel på kommunikationsinstrument som ingår i standardmässiga *UNIX*-system. I denna uppgift ska vi titta på en variant av en pipe, som kan finnas som en fil i filhierarkin: en så kallad *fifo*, det är väsenligen en pipe, *fifo* står för *"first-in-first-out"*. Vi skapar en fifo genom följande kommandoradsargument:

```
$ mkfifo pipe
```

och om det går att genomföra så uppstår en så kallad fifo som här kallas "pipe" i filhierarkin. Det finns dock en liten hake med den här uppgiften och det är att fifon måste skapas lokalt, alltså under hemkatalogen, det går inte att skapa den i katalogen där du ska lämna in dina svar (alltså H). Se först till att du stå i din hemkatalog, inte i H (där du senare ska lämna in svaret på denna uppgift). Utför detta kommando i en kommandoruta nu och gör sedan `ls -l`, så kommer du att se att du skapat en pipe. Resultatet av `ls -l pipe` på min maskin blev följande:

```
prw-r--r-- 1 johnny johnny 0 Jun  3 22:37 pipe
```

där ett `p` längst till vänster anger att vi skapat en fifo. (*p* står för *pipe* som är en synonym till *fifo*.)

Vi ska nu använda den här fifon för att kommunicera med en demonliknande process, vi börjar med att införa ett program som skapar en process som läser från en fifo (programmet finns i filerna som hör till tentan, programmet heter `readfromfifo`):

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
main()
{
  char str[10], *res; int x;
  FILE *fd=fopen("./pipe","r");
  while(1) {
    res=fgets(str,10,fd);
    if(res!=NULL) {
      x=atoi(str); printf("x: %d.\n", x);
      system("ps -e -o pid,ppid,comm | grep uppg1");
    }
    sleep(1);
  }
}
```

Som vi ser använder vi standardio-strömmar (`FILE *`) för att läsa från fifon. Kör detta program och ge kommandot

```
$ echo "3">pipe
```

programmet `readfromfifo` ska då kunna fånga upp och skriva ut den 3:a som du skrev in.

Uppgift: skriv om `readfromfifo` så att den lägger sig som ett barn under `init`, och tar emot heltal av användaren genom fifon och gör följande: om ett positivt heltal matas in (1,2,3 osv) så ska så många barnprocesser som anges av heltalet skapas under `init`, dessa barnprocesser ska bara sova i 15 sekunder därefter avslutas. Vidare ska också ett anrop till `system("ps -e -o pid,ppid,comm | grep uppg1")` ske så att vi klart kan se de nyskapade processerna. Observera att du behöver döpa om det körbara programmet till `uppg1` för att `system("ps -e -o pid,ppid,comm | grep uppg1")` ska fungera. Om användaren skickar 0 eller något negativt tal ska programmet avslutas. En testkörning av programmet ser ut så här:

```
$ ./uppg1
$ ps -e -o pid,ppid,comm | grep uppg1
 2251     1 uppg1
$ echo "2">pipe
$ Adding 2 children under init.
 2251     1 uppg1
 2255     1 uppg1
 2256     1 uppg1
```

Här väntar vi 15 sekunder så att barnen hinner avslutas och sedan fortsätter vi och ger kommandot `ps -e -o pid,ppid,comm | grep uppg1` för att se att barnen avslutats. (Vi ger alltså det kommandot både i programmet som löser uppgiften och vi använder det när vi provkör för att se att det fungerar som det ska.)

```
$ ps -e -o pid,ppid,comm | grep uppg1
2251     1 uppg1
```

Bra! Inga barn kvar. Då fortsätter vi och skapar tre nya barn.

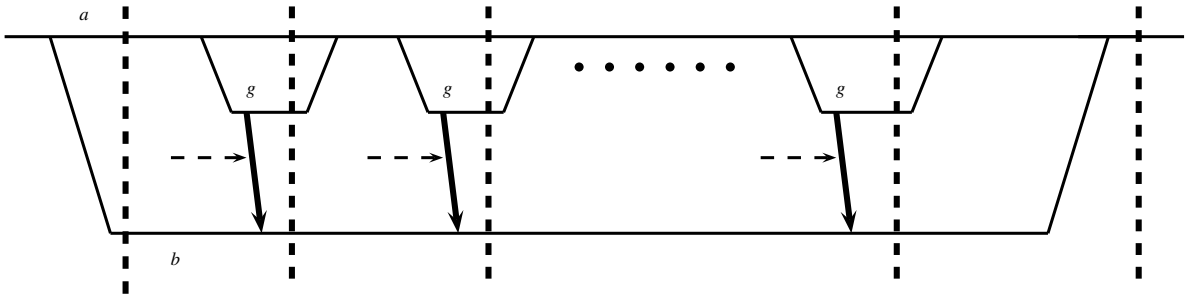
```
$ echo "3">pipe
$ Adding 3 children under init.
 2251     1 uppg1
 2263     1 uppg1
 2264     1 uppg1
 2265     1 uppg1
$
```

Vi avslutar med

```
$ echo "0">pipe
$ Stopping daemon.
```

där utskriften `Stopping daemon` var det sista som kom från vårt program. Lägg märke till att eftersom vi har en demon som kör så synkroniseras inte dess utskrifter med kommandoradsprompten (`$`), vi anger alltså inte `Stopping daemon.` som kommando, det är en utskrift från demonen, men det ser ut som om det är något som användaren skriver in, men det är det alltså inte. (Och egentligen kör vi inte en demon i strikt mening, en riktig demon gör ju inga utskrifter, men vi kallar den för en demon eftersom den ligger som barn under init.)

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



Processen `a` är föräldraprocessen som ska skapa barn av två slag, dels den process som benämns `b` och ett visst antal av de processer som benämns `g` i tidsdiagrammet ovan. Antal processer som benämns `g` kan variera och bestäms av ett kommandoradsargument men vi ska alltid ha en process som benämns `b`. I tidsdiagrammet ovan ser vi också att `g`-processerna ska kommunicera med `b`-processen, det finns pilar som leder från `g`:na till `b`. Dessa pilar ska indikera att det finns *en* pipe som leder från `g`:na till `b` - normalt brukar ju en pil vara en pipe men här har vi alltså flera pilar som ska indikera en pipe. Vi visar detta särskilt också genom att alla dessa pilar pekas på med streckade pilar. I denna uppgift räcker det alltså med en pipe.

Men vi ska som sagt ha flera `g`-processer, "`g`" står för vad processen gör, den ger sitt processid på sin standardout, den ska se ut så här:

```
#include <unistd.h>
main() int pid=getpid(); write(1,&pid,sizeof(int));
```

Denna kod får alltå *inte* ändras. Hela programmet ska startas på följande sätt:

```
$ ./uppg2 3
```

där 3:an indikerar hur många g-processer som ska startas. De tjocka streckade lodräta linjerna ska ett anrop ske av slaget

```
system("ps -o pid,ppid,comm");
```

så att vi kan få en rapport över vilka processer som skapats och deras relationer.

Vad ska programmet göra då? Ja, som vi ser så skriver g-processerna ut sina processid:n på standard-out, av tidsdiagrammet framgå att b-processen ska ta emot dessa processid:n och det som b-processen ska göra är att summera dessa processid:n. Fullständigt meningslöst att summera processid:n, men vi kan se det som en modell av något annat, b-processen ska summera alla processid:n som sagt och innan den avslutar ska den skriva ut denna summa på skärmen. Om ni vill så får ni gärna skapa en till pipe som b-processen skriver resultatet i till a-processen och så får a-processen sköta utskriften, det skulle kunna anses vara bättre för då blir all kommunikation med användaren samlad i a-processen, men gör det som ni får att fungera.

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

```
$ ./uppg2 2
  PID  PPID COMMAND
 1618  1614 bash
 1674  1618 uppg2
 1675  1674 uppg2
 1676  1674 sh
 1677  1676 ps
  PID  PPID COMMAND
 1618  1614 bash
 1674  1618 uppg2
 1675  1674 uppg2
 1678  1674 uppg2
 1679  1674 sh
 1680  1679 ps
  PID  PPID COMMAND
 1618  1614 bash
 1674  1618 uppg2
 1675  1674 uppg2
 1681  1674 uppg2
 1682  1674 sh
 1683  1682 ps
Sum of pids: 3359.
  PID  PPID COMMAND
 1618  1614 bash
 1674  1618 uppg2
 1684  1674 sh
 1685  1684 ps
```

I testkörningen till vänster ser vi att det skapas två stycken g-barnprocesser med respektive processid 1678 och 1681. Summan av dessa tal är 3359 och det skrivs ut som vi ser.

g-barnprocesserna kör efter varandra och inte samtidigt som också framgår av tidsdiagrammet och test-körningen.

(3) Nedanstående program som involverar två trådar kan råka ut för deadlock. Gör modifikationer i programmet så att deadlock helt säkert inte kan uppstå. Skriv också en förklaring *i din programkod* som motiverar dina ändringar och hänvisa till något av de tre nödvändiga villkoren för deadlock och ange vilket av dessa villkor du upphäver och motivera därigenom dina modifkationer av den ursprungliga programkoden. *(Programmet finns förstås som fil tillsammans med de andra filerna som hör till tentan.)*

Nu uppkommer ju förstås frågan om hur mycket man får ändra i programmet så att deadlock säkert inte uppstår, om vi tar bort allting och bara skriver in något annat som inte har med trådar alls att göra så kan ju inget deadlock uppstå, men det skulle vara en alltför förenklad lösning. För att undvika att förenkla för mycket så får vi tänka så här: i programmet är det tänkt att två trådar ska slumpvis välja mellan 5 resurser och låsa två av dem. Denna procedur, att två trådar låser varsinna två slumpmässigt valda resurser bland 5 möjliga, måste bevaras i din lösning. Ett annat sätt att se detta är att du som student i denna tentamensuppgift ska demonstrera att du behärskar att undvika deadlock i trådprogrammering, om du verkligen kan det så kan du ta fram en lösning på uppgiften som inte förenklar för mycket, det är kravet på en godkänd lösning.

Alldeles till höger om programmet finns en testkörning som visar en deadlock. Och längst till höger finns en testkörning som visar en provkörning av en fungerande rättad variant.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

pthread_mutex_t resource[5];
char res_str[] = "00000";

int pick(){return rand()%5;}
int pick2(){return rand()%2+1;}

void* t1 (void * p) {
  int r1, r2;
  while(1) {
    r1=pick(); r2=pick();
    sleep(pick2());
    pthread_mutex_lock(&resource[r1]);
    res_str[r1]='1';
    pthread_mutex_lock(&resource[r2]);
    res_str[r2]='1';
    sleep(pick2());
    res_str[r1]='0';
    res_str[r2]='0';
    pthread_mutex_unlock(&resource[r1]);
    pthread_mutex_unlock(&resource[r2]);
  }
}

void* t2 (void * p) {
  int r1, r2;
  while(1) {
    r1=pick(); r2=pick();
    sleep(pick2());
    pthread_mutex_lock(&resource[r1]);
    res_str[r1]='2';
    pthread_mutex_lock(&resource[r2]);
    res_str[r2]='2';
    sleep(pick2());
    res_str[r1]='0';
    res_str[r2]='0';
    pthread_mutex_unlock(&resource[r1]);
    pthread_mutex_unlock(&resource[r2]);
  }
}

main() {
  int i=0,n;
  pthread_t id1, id2; srand(time(0));
  pthread_create(&id1,NULL,&t1,NULL);
  pthread_create(&id2,NULL,&t2,NULL);
  while(i++<30)
  {
    sleep(1);
    printf("%2d: %s \n", i, res_str);
  }
}
```

```
$ ./uppg3
 1: 00000
 2: 12020
 3: 10000
 4: 10010
 5: 00202
 6: 00202
 7: 10100
 8: 22000
 9: 22000
10: 01000
11: 01202
12: 01000
13: 01000
14: 01220
15: 01220
16: 01000
17: 01200
18: 01200
19: 01200
20: 01200
21: 01200
22: 01200
23: 01200
24: 01200
25: 01200
26: 01200
27: 01200
28: 01200
29: 01200
30: 01200
```

Deadlock syns på de sista raderna

```
$ ./uppg3c
 1: 00000
 2: 12120
 3: 02020
 4: 00000
 5: 01100
 6: 00220
 7: 00220
 8: 00110
 9: 00000
10: 12021
11: 02020
12: 11000
13: 00220
14: 00221
15: 00101
16: 00000
17: 00022
18: 00022
19: 00011
20: 02020
21: 00000
22: 00011
23: 00202
24: 01100
25: 20002
26: 00000
27: 01100
28: 22000
29: 22000
30: 10001
```

Hela körningen är fri från deadlock.