UPPGIFTER AV FÖRESLAGEN TENTAMENSKARAKTÄR, LÅTSASTENTA DEN 27 FEBRUARI 2017

Inför datortentan III

Detta är tredje dokumentet i en följd av dokument som med syfte att klargöra kursmål och examinationsformen på datortentan. I det här dokumentet presenteras förslag på uppgifter som vi kanske kan anse är av tentamenskaraktär.

Tolkning av ingenjörsmässiga kurs- och examensmål och motivation av denna examinationsform

Examinationen i kurserna behöver befrämja studenters uppfyllande av målen med utbildningen och den specifika kursen, men det här måste konkretiseras varför vi gör en uppräkning av kursens systemprogrammeringsmässiga huvudmoment. På datortentan ska studenten

1. ... uppvisa förmåga att skapa och/eller hantera flerprocessiga program där de ingående processerna kommunicerar med *UNIX* standardmässiga paradigm med fildeskriptorer - det här med `close()`, `dup()`, `fork()`, `exec()` osv. Detta kallas *IPC*, InterProcess Communication. Särskild vikt läggs vid mekanismen att `close()` avslutar en kommunikation. Härvid används olika kommunikationsinstrument som sockets och pipes.

2. ... uppvisa förmåga att skapa och/eller hantera flerprocessiga program där de ingående processerna ges olika roller och intar olika relationer till varandra. (Barn, barnbarn, syskon, kusiner etc.) I synnerhet krävs att studenten ska kunna skapa/hantera så kallade demonprocesser och även undvika/hantera så kallade zombieprocesser.

3. ... uppvisa förmåga att skapa och/eller hantera flertrådade program/processer och därvid uppvisa medvetenhet om problematik som *kapplöpning* och *deadlock*. Studenten ska kunna skapa/hantera flertrådade program som undviker dessa båda problem. Så kallade *p*-trådar (*POSIX-threads*) kommer att användas.

Dessa tre punkter utgör en tolkning av vad som krävs och studenten ska uppfylla alla tre krav parallellt, det betyder att uppgifterna som ges på datortentan kan befatta sig med aspekter av två eller rent av alla tre krav samtidigt. Till exmpel kan alltså en problemställning ges där trådar (mål 3) ska kommunicera med via en pipe eller en socket och göra detta på ett solitt sätt, dvs avsluta med `close()` så att principen om solid *IPC* upprätthålls (mål 1).

Tillåtna hjälpmedel

Programmering är ju en aktivitet som normalt innebär att programmeraren har tillgång till olika hjälpmedel. De hjälpmedel som ges på datortentan blir följande:

1. Kurslitteraturen, *Advanced Linux Programming*, *Beej's Guide to Network Programming*.
2. Exempelkod där systemanropen används för att skapa program med inslagen från de tre kraven ovan. (Denna lista på exempelprogram tar vi fram under kursens gång.)
3. En körande *UNIX*-miljö vari tentamensuppgifterna ska lösas. Miljön i sig innehåller också hjälpmedel som manualsidor och systemprogrammeringsverktyg.

Svarsformat

För att se att de program/lösningar som lämnas in behvöver studenten placera anrop till `ps` respektive `sleep`. Detta görs med fördel med systemanropet `system()` och eftersom detta fungerar så väl så kommer det att vara ett krav att programmet utformas med välplacerade anrop till detta systemanrop samt till `sleep()`.

Anropet till `system/ps` tar då formen

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

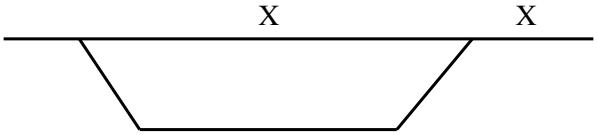
medan anropet till `sleep()` kan göras direkt i *C*-koden utan att gå via `system()`. Det kommer att vara ett krav att studenten gör detta anrop på kritiska tidpunkte i tidsdiagrammen. Dessa punkter i programmet kommer att markeras med ett kryss.

Detta illustreras i följande exempel. Barnet i koden

```
if(!fork())
{
  sleep(1);
  exit(0);
}
system("ps -o pid,ppid,pgid,sess,comm"); //anrop 1
wait(0);
system("ps -o pid,ppid,pgid,sess,comm"); //anrop 2
```

kommer att synas säkert wid anrop 1 ovan. I anrop 2 så kommer inte barnprocessen att synas eftersom den avslutats då med `exit(0)` och eftersom föräldern gör `wait` innan anrop 2 så kommer inte barnet att synas vid anrop 2. Om vi inte har `sleep(1)` så kan det hända att barnprocessen inte hinner skapas innan föräldern gör anrop 1 så då syns inte barnet i vare sig anrop 1 eller anrop 2.

Om vi ser på motsvarande tidsdiagram har det följande utseende



Observera särskilt de två kryssen som alltså symboliserar anropet till system/ps. Det kommer som sagt att vara ett särskilt krav att dessa anrop görs i de positioner som anges, och att utskriften har det principiella resultat som anges av de testutskrifter som också ges i samband med formuleringen av tentauppgifterna.

Här är en provkörning av ovanstående kod:

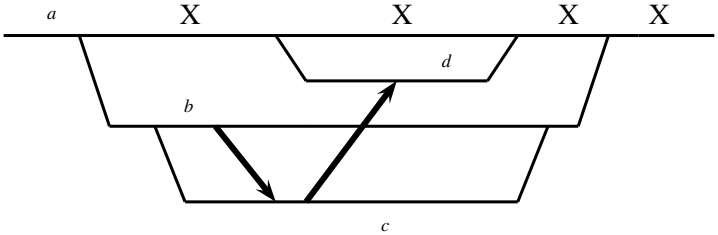
```
  PID  PPID  PGID  SESS COMMAND
 4399  3164  4399  4399 bash
 4466  4399  4466  4399 a.out
 4467  4466  4466  4399 a.out
 4468  4466  4466  4399 sh
 4469  4468  4466  4399 ps
  PID  PPID  PGID  SESS COMMAND
 4399  3164  4399  4399 bash
 4466  4399  4466  4399 a.out
 4470  4466  4466  4399 sh
 4471  4470  4466  4399 ps
```

som vi ser syns barnet som tydligen har processid 4467 vid första utskriften som kommer i respons till första aropet. Vid andra anropet är barnet borta.

Tre uppgifter av föreslagen tentamenskaraktär

Nedan följer uppgifter som gavs vid en låtsastenta den 27 februari 2017- Syftet med detta var att få en preliminär uppfattning om hur denna examinationsform fungerar. Tre studenter från andra årskursen arbetade med uppgifterna under 2 timmar. På den riktiga datortentan ges 5 timmar. Uppgifternas formuleringar har utvidgats och förtydligats i respons till den utvärdering som följde. En film kommer att finns publicerad som visar hur man kan arbeta med att lösa problem av detta slag i en steg-för-stegprocedur. (I skrivande stund är inte filmen klar, men den kommer snart.)

Uppgift 1. Skriv ett flerprocessigt program som beskrivs av nedanstående tidsdiagram:



Föräldraprocessen *ska* göra totalt 4 anrop till system/ps och resultatet av den utskriften ska ha ha fljande principiella utseende:

Koden till processen *d ska* vara följande (include-direktiv utelämnade)

```
main()
{
  int p;
  while(read(0,&p,sizeof(int)))printf("p = %d.\n", p);
}
```

och d ska initieras med ett anrop till `execlp` av typen `execlp("./d","./d",NULL);` i sin föräldraprocess (a). (Vi utelämnar i detta tidsdiagram relationen till det omgivande skalet.) Samtliga systemanrop ska kontrolleras, förutom till `fork()`, `system()` och `sleep()`.

De tjocka pilarna representerar pipes och det som ska ske är att b ska skriva sitt processid i pipen till c, därefter ska c skicka vidare detta processid till d. Men då detta r̈ gjort ska b även skicka c's processidid och c ska vidarebefodra detta till d Processen c ska fungera i form av en loop som kan ta emot ett godtycligt antal tal och vidarebefordra till d. Avslutning med stängningar ska indikera att kommunikationen är klar. Som vanligt ska välplacerade anrop till

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

och `sleep()` användas för att se att programmet fungerar som det ska.

Uppgift 2.

Studera nedanstående program:

```
/*
** seed to a deamon. Demonstrates unix sockets
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <signal.h>
#include <wait.h>

#define SOCK_PATH "echo_socket"

void sigchld_handler(int s)
{
  while(waitpid(-1, NULL, WNOHANG) > 0);
}

int main(void)
{
  int s, s2, t, len;
  struct sockaddr_un local, remote;
  struct sigaction sa;

  sa.sa_handler = sigchld_handler; // reap all dead processes
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    perror("sigaction");
    exit(1);
  }

  if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }
  if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
```

```
    perror("socket");
    exit(1);
  }

  local.sun_family = AF_UNIX;
  strcpy(local.sun_path, SOCK_PATH);
  unlink(local.sun_path);
  len = strlen(local.sun_path) + sizeof(local.sun_family);
  if (bind(s, (struct sockaddr *)&local, len) == -1) {
    perror("bind");
    exit(1);
  }

  if (listen(s, 5) == -1) {
    perror("listen");
    exit(1);
  }

  for(;;) {
    int done, n;
    printf("Waiting for a connection...\n");
    t = sizeof(remote);
    if ((s2 = accept(s, (struct sockaddr *)&remote, &t)) == -1) {
      perror("accept");
      exit(1);
    }

    //Work with s2

  }
  return 0;
}
```

Programmet finns bland exempelkoderna som följer med denna tenta. Det finns också ett enkelt klientprogram.

Uppgiften är att omvandla detta program till en server som kör som barn under *init*. Det ska bli en server som är flerprocessig och som gör följande:

1. Vid start skapar den en socket i `/tmp/` och det är där som klienter ansluter. Klienten skicka då sitt processid så att servern vet vilken process som ska ha signalen.
2. Vid ett anrop skapas en parallell process som väntar i 20 sekunder och därefter skickas signalen `KILL` till den process som anropat servern. Flera anrop till servern ska kunna ske parallellt och servern ska skapa en barnprocess i respons till samtliga anrop.
3. Då en klient tar emot signalen `KILL` så ska servern som sista sak göra ett anrop till system/ps för att kontroller att processen tagits bort. (För att se att programmet fungerar som det ska så kan mer anrop till system/ps och sleep vara nödvändiga.

Uppgift 3

Studera nedanstående program:

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

#define SLEEP 1
#define WANT 2

int run = 1;
```

```
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m3 = PTHREAD_MUTEX_INITIALIZER;

char state[] = "r1   t1   r2   t2   r3   t3   r1";

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

void* tf1 (void*p) {
  while(run)
  {
    switch(choose(2))
    {
      case SLEEP: sleep(2*choose(3)); break;
      case WANT:
        state[3]='w'; pthread_mutex_lock(&m1); state[3]='-';
        state[8]='w'; pthread_mutex_lock(&m2); state[8]='-';
        state[5]='T'; sleep(2*choose(5));
        pthread_mutex_unlock(&m1); pthread_mutex_unlock(&m2);
        state[3]=' '; state[8]=' '; state[5]='t';
        break;
    }
  }
  return NULL;
}

void* tf2 (void*p) {
  while(run)
  {
    switch(choose(2))
    {
      case SLEEP: sleep(2*choose(3)); break;
      case WANT:
        state[13] = 'w'; pthread_mutex_lock(&m2); state[13] = '-';
        state[18] = 'w'; pthread_mutex_lock(&m3); state[18] = '-';
        state[15]='T'; sleep(2*choose(5));
        pthread_mutex_unlock(&m2); pthread_mutex_unlock(&m3);
        state[13]=' '; state[18]=' '; state[15]='t';
        break;
    }
  }
  return NULL;
}

void* tf3 (void*p) {
  while(run)
  {
    switch(choose(2))
    {
      case SLEEP: sleep(2*choose(3)); break;
      case WANT:
        state[23]='w'; pthread_mutex_lock(&m3); state[28]='-';
        state[28]='w'; pthread_mutex_lock(&m1); state[23]='-';
        state[25]='T'; sleep(2*choose(5));
        pthread_mutex_unlock(&m1); pthread_mutex_unlock(&m3);
        state[23]=' '; state[28]=' '; state[25]='t';
        break;
    }
  }
```

```
  return NULL;
}

int main (int argc, char* argv[]) {
  pthread_t t1, t2, t3; int round = 0;

  srand(time(0));
  pthread_create(&t1,NULL,&tf1,NULL);
  pthread_create(&t2,NULL,&tf2,NULL);
  pthread_create(&t3,NULL,&tf3,NULL);

  while(round<24) {
     printf("Round %2d: %s\n", round+1, state);
     sleep(2);
     round++;
  }
  run = 0;

  pthread_join(t1,NULL); pthread_join(t2,NULL);
  pthread_join(t3,NULL);
  return 0;
}
```

Det är en enklare version av laboration 3 där tre trådar samsas om tre resurser och vi ska ställa frågor om deadlock här. Det kompileras med kommandot `gcc threads.c -o t -pthread`. En testkörning ser ut så här:

```
$ ./t
Round  1: r1   t1   r2   t2   r3 - T3 w r1
Round  2: r1 w t1   r2 - t2 w r3 - T3 w r1
Round  3: r1 w t1   r2 - t2 w r3 - T3 w r1
Round  4: r1 w t1   r2 - t2 w r3 - T3 w r1
Round  5: r1 w t1   r2 - t2 w r3 - T3 w r1
Round  6: r1 w t1   r2 - t2 w r3 - T3 w r1
Round  7: r1 w t1   r2 - t2 w r3 - T3 w r1
Round  8: r1 w t1   r2 - t2 w r3 - T3 w r1
Round  9: r1 - t1 w r2 - T2 - r3   t3   r1
Round 10: r1 - T1 - r2   t2   r3   t3   r1
Round 11: r1 - T1 - r2   t2   r3   t3   r1
Round 12: r1 - T1 - r2   t2   r3   t3   r1
Round 13: r1 - T1 - r2   t2   r3   t3   r1
Round 14: r1 - T1 - r2   t2   r3 w t3 w r1
Round 15: r1 - T1 - r2   t2   r3 w t3 w r1
Round 16: r1 - T1 - r2   t2   r3 w t3 w r1
Round 17: r1 - T1 - r2   t2   r3 w t3 w r1
Round 18: r1 - T1 - r2   t2   r3 w t3 w r1
Round 19: r1 - T1 - r2   t2   r3 w t3 w r1
Round 20: r1 - T1 - r2   t2   r3 w t3 w r1
Round 21: r1   t1   r2 - t2 w r3 - T3 w r1
Round 22: r1   t1   r2 - t2 w r3 - T3 w r1
Round 23: r1 w t1   r2 - t2 w r3 - T3 w r1
Round 24: r1 w t1   r2 - t2 w r3 - T3 w r1
```

Tre trådar som sagt som delar på tre resurser, trådarna kan vara i två tillstånd, 1: sovandes (`SLEEP` ovan) och 2: vilja ha resurser (`WANT` ovan). Slumpfunktionen `choose()` skapar slumptal som gör att trådarna gör olika saker. Programmets progression beskrivs grafiskt, likt laboration 3, men lite enklare, en rad, till exempel nr 23 ovan:

```
Round 23: r1 w t1   r2 - t2 w r3 - T3 w r1
```

betyder att tråd 1 (som kodas med `t1`) vill ha resurs `r1`, den väntar på denna alltså, tråd 2 (`t2`) äger `r2`, det symboliseras av ett streck mellan den grafiska representationen för `r2` och `t2`. På samma sätt ser vi att tråd 3

äger `r3` och väntar på `t1`. Egentligen ska `t3` äga `r1` men programmet är inte riktigt stabilt. Det kommer dock inte att ha så stor betydelse för svaret på fr.

1. Kan deadlock uppstå i detta program? Varför varför inte?
2. Om deadlock kan uppstå, skriv om programmet så att deadlock säkert inte uppstår. ändra så lite som mjligt i koden.
3. Skriv sedan om programmet så att deadlock säkert uppstår genom att återigen ändra så lite som möjligt i koden.

Svara i två separata programfiler där du i ena filen skriver in svaret på delfåga 1.