Tentamen i Operativsystem, HI1025:TEN1 - 9 juni, 2016

Allmänna instruktioner. Tentamen innehåller 10 uppgifter med totalt 49 poäng. För lägsta godkända betyg (E) krävs ungefär 24 poäng. För att få komplettera (Fx) krävs ungefär 22 poäng.
Viktigt: När du svarar på en fråga ska det i allmänhet framgå varför du vet svaret frågan, det uppnås enklast genom att du sätter in termer och begrepp som du hanterar i deras sammanhang på ett korrekt sätt. Men det är också viktigt att inte bli för mångordig, i uppgifter med 1 poäng krävs ett kort svar, i uppgifter med mer poäng behöver svaret utvecklas mer. **OBS: Svara *inte* i tentan, *bara* på svarspapper.** Av layoutskäl saknas nödvändiga `#include`-direktiv i källkoden i vissa av uppgifterna, men uppgifterna ska behandlas som om direktiven fanns där. *Lycka till!*

Uppgifter

(1) I ett modernt *UNIX*-system är filer organiserade i en filhierarki vars underliggande relationer till systemets lagringsmedium specificeras av filen `/etc/fstab` (alltså tex hur katalognamn hänger ihop med hårddiskar). Beskriv innehållet i denna fil och ange hur innehållet kan specifiera det den gör. Din redogörelse ska innehålla orden *montering*, *filsystem*, *partition/enhet*. Det ska klart framgå från din beskrivning att detta innebär en flexibilitet och du ska motivera den flexibiliteten i ditt svar. (Du behöver inte ange exakt syntax för innehållet i `/etc/fstab`). **(4p)**

(2) Förklara vad en demonprocess är och ange två egenskaper som är särskilt viktiga hos en demonprocess. Förklara varför servrar (i klient-serverapplikationer) gärna uppträder i form av demonprocesser och hänvisa till de två egenskaperna. (Typ ”demoner är ... serverprocesser är ofta demoner för att ... bla bla bla egenskap 1 ... för då ... och sen egenskap 2 som är bra för att ...”.) **(5p)**

(3) Låt `B` beteckna en process som någon annan skrivit koden till. Processen `B` läser data från standard-in och skriver ett resultat på standard-ut (som de flesta standardmässiga *UNIX*-processer gör).

Vi vill skriva programmet `A` som ska vara en så kallad wrapper till `B` på följande sätt: en användare ska kommunicera med `B` *via* `A`. (`A` kanske i sin tur kommunicerar med `B` i form av pipes.)

Beskriv hur vi uppnår det genom att använda omdirigering vid Inter-Process Communication. Din redogörelse ska innehålla beskrivningar av `dup()`, `fork()`, `exec()`, och beskriva begreppet ”omdirigering” med formuleringar involverande fildeskriptortabellerna och stdin/stdout. **(6p)**

(4) Vid installation av operativsystem i allmänhet förekommer fyra allmänna delsteg: *Formatering*, *Konfiguration*, *Partitionering*, *Kopiering/kompilering av programvara*. Förklara dessa delsteg och ange i vilken ordning de kommer vid en installation. **(5p)**

(5) Förklara innebörden av vad följande typer av processer är och ge enkla exempel på hur dessa processer uppstår: *(a) Barnprocess*, *(b) Zombieprocess*. **(4p)**

(6) IPC: *(a)* När uppkommer en så kallad *broken pipe*? **(1p)** *(b)* Varför bör man läsa från en *pipe* direkt i anslutning till att man skriver till den? **(1p)** *(c)* BSD-sockets (sådana vi studerat i OS-kursen) skiljer sig väsentligt från *pipes* på flera sätt. Beskriv två av dessa skillnader. **(2p)**

(7) Redogör för de tre nödvändiga villkoren för att låsning (*deadlock*) i ett system med parallella processer/trådar med delade resurser ska kunna uppkomma. Beskriv också två olika strategier för att hantera problemet med låsning som inriktar sig på två av dessa olika villkor. **(5p)**

(8) Beskriv de tre CPU-schemaläggningsstrategierna *FIFO (FCFS) - First In First Out* , *SJF - Shortest Job First*, *RR - Round Robin*. Ange för- och nackdelar med var och en av dessa strategier.

(*Ledning:* Dessa för- och nackdelar kan knytas till jobbtyper: vissa typer av jobb lämpar sig för vissa CPU-schemaläggningsstrategier och vissa andra typer av jobb mår bättre av andra CPU-schemaläggningsstrategier. Motiveringar är som vanligt självklart nödvändiga.) **(6p)**

(9) Studera nedanstående program:

```
c(){if(!fork()){sleep(1); exit(0);} else wait(0);}

main(){
  if(!fork()){ c(); c(); exit(0);}
  c(); c(); c();
  wait(0);
}
```

(a) Rita ett tidsdiagram som illustrerar släktförhållandena mellan de ingående processerna. **(4p)**

(b) Hur många processer är maximalt igång till följd av att programmet kör? Motivera ditt svar genom att hänvisa till tidsdiagrammet. **(2p)**

(10) Studera nedanstående program:

```
int i, run = 1, sleep_time, counter = 0; pthread_t thr[2];
pthread_mutex_t counter_guard = PTHREAD_MUTEX_INITIALIZER;
void * incr (void * p)
{
  int *id;
  id = (int*)(p);
  printf("id: %d.\n", *id);
  sleep(1);
  while(run)
  {
    pthread_mutex_lock(&counter_guard);
    counter++; printf("%d: %d.\n", *id, counter);
    pthread_mutex_unlock(&counter_guard);
    //Tråd 2 ska sova längre än tråd 1 därför har vi lagt in multiplikation
    //med *id nedan (i det här programmet blir *id 1 eller 2):
    sleep_time=(rand()%2)*(*id)+1;
    sleep(sleep_time);
  };
  pthread_exit(0);
}
main()
{
  srand(time(0)); int id[2] = {1,2};
  for(i=0;i<2;i++)pthread_create(&thr[i],NULL,&incr,&id[i]);
  sleep(10); run = 0;
  for(i=0;i<2;i++)pthread_join(thr[i],NULL);
}
```

Programmet har följande körning:

```
id: 1.
id: 2.
1: 1.
2: 2.
1: 3.
1: 4.
2: 5.
2: 6.
1: 7.
2: 8.
1: 9.
1: 10.
2: 11.
```

Alltså två trådar som ökar på en gemensam variabel, `counter`. Programmet är dock inte trådsäkert. En kapplöpning (*race condition*) skulle kunna uppstå någonstans. Beskriv denna kapplöpning (motivera hur den skulle kunna uppstå). Skulle denna kapplöpning kunna leda till problem? Motivera. Föreslå en förbättring för att avhjälpa problemet. **(6p)**