# Tentamen i Operativsystem, HI1025:TEN1 - 23 maj, 2014

**Allmänna instruktioner.** Tentamen innehåller 10 frågor med totalt 48 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.**

**Anmärkning:** Av layoutskäl saknas nödvändiga #include-direktiv i källkoden i vissa av uppgifterna, men uppgifterna ska behandlas som om direktiven fanns där.

**Lycka till!**
*Johnny*

## Uppgifter

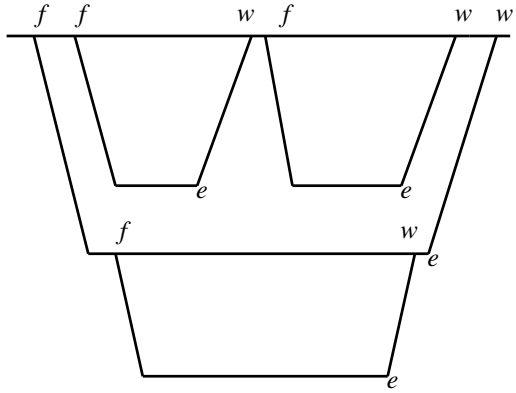
(1) Antag att vi har ett antal parallella processer som turas om om ett antal resurser. Kan vi då ha *deadlock utan mutual exclusion*? Motivera.
**(2p)**

(2) Förklara skillnaden mellan så kallad *mikrokärna* och *monolitisk kärna*. Ange fördelar med båda konstruktionerna. (Motivera dina svar, som alltid.)
**(4p)**

(3) Förklara följande begrepp
(a) *Processbild / Process image*
(b) *Strictly Pre-emptive* (gällande *time sharing* mellan processer)
(c) *Politely Pre-emptive* (gällande *time sharing* mellan processer)
(d) *Context switch*
**(8p)**

(4) Diskutera för- och nackdelar med stora och små kluster i filsystem, det ska klart framgå varför ett filsystem blir långsammare och rymligare i ena fallet men snabbare och mindre rymligt i andra fallet.
**(4p)**

(5) Studera nedanstående tidsdiagram över ett antal parallellt körande processer:




Bokstäverna *f*, *e* och *w* betyder förstås **fork**, **exit** respektive **wait**. Skriv *C*-kod som skapar denna principiella konfiguration av körande parallella processer. Du behöver inte deklarera några variabler eller ange några inkluderingsdirektiv men anropen till **fork**, **exit** respektive **wait** måste vara helt korrekta, alltså komma i rätt ordning, för att skapa den önskade konfigurationen.
**(5p)**

(6) Laboration 1 i kursen bestod i att skapa en demonbaserad server som kunde ta emot klientanslutningar via en socket. Vi använde **fork** för att skapa parallella skeenden i demonen för att möjliggöra att hantera flera klienter parallellt. Det finns dock en annan variant av **fork** som heter **vfork**. Studera följande utdrag ur manualsidan till **vfork**:

**VFORK(2) Linux Programmer's Manual VFORK(2)**

NAME vfork - create a child process and block parent

...

**Linux Description**

vfork(), just like fork(2), creates a child process of the calling process. For details and return value and errors, see fork(2).

vfork() is a special case of clone(2). It is used to create new processes without copying the page tables of the parent process. It may be useful in performance-sensitive applications where a child is created which then immediately issues an execve(2).

vfork() differs from fork(2) in that the parent is suspended until the child terminates (either normally, by calling _exit(2), or abnormally, after delivery of a fatal signal), or it makes a call to execve(2). Until that point, the child shares all memory with its parent, including the stack. The child must not return from the current function or call exit(3), but may call _exit(2).

Signal handlers are inherited, but not shared. Signals to the parent arrive after the child releases the parent's memory (i.e., after the child terminates or calls execve(2)).

Beskriv hur du skulle ändra i designen på laboration 1 för att få laborationen att fungerar med **vfork** istället för **fork**. Är denna variant bättre eller sämre än den ursprungliga versionen av laboration 1? Motivera.
*Ledning:* Det är särskilt viktigt att fundera på hur du får servern att bibehålla förmågan att hantera klienter parallellt. Kommandot **execve** ligger bakom alla kommandona **execlp**, **execvp**, **execl**, dvs, alla kommandon av typen **execlp**, **execvp**, **execl** innebär ett anrop till **execve**.
**(5p)**

(7) Rotpartitionen, /, i ett körande *UNIX*-system har blivit full. För att reparera detta har systemadministratören skaffat en ny hårddisk och hen vill koppla in den i det nya systemet. För att göra detta behöver hen välja en monteringspunkt (katalog) i den partition som blivit full där en ny partition från den nya hårddisken ska monteras. Systemadministratören har bestämt sig för att denna punkt ska vara /var, dvs alla filer i /var ska över på den nya hårddisken (på någon partition) och filen /etc/fstab ska uppdateras. I /etc/fstab står det, innan denna process, följande:

```
/dev/sda1   /
/dev/sda2   /tmp
/dev/sda3   /home
```

det vill säga endast tmp och home är på separata partitioner.

Ange de kommandon som systemadministratören behöver utföra för att installera den nya hårddisken i det körande systemet och ange hur /etc/fstab ska uppdateras för att systemet ska köra problemfritt med den nya hårdisken installerad. Det sista kommandot ska vara **reboot** och då ska alla nödvändiga modifikationer vara gjorda (beskrivna som kommandon) som gör att systemet kan komma igång igen.
Vi antar att partitionen på den nya hårddisken som ska innehålla filerna i /var heter /dev/sdb1 och att den är formaterad till något lämpligt filsystem. Du behöver inte fundera över dess typ.

*Ledning:* Växeln -a på kommandot cp är arkiveringsfunktionen på cp-kommandot som gör så att cp fungerar rekursivt och bevarar alla filers attribut. Innehållet i /etc/fstab är också noterat på ett förenklat sätt, vi anger bara partition följt av monteringspunkt, det är förstås inte rätt syntax, men vi använder detta sätt att skriva här i denna uppgift så belastas inte studenten med krav på att komma ihåg exakt hur man skriver en fstab-fil.
**(6p)**

(8) Vad är problemet med så kallade *Zombier*? Skulle det vara möjligt att lösa detta problem genom att skriva en demon som hela tiden ligger och lurpassar på alla processer i ett körande *UNIX*-system och som så fort den upptäcker en zombie helt enkelt dödar den processen? Varför? Varför inte?
**(4p)**

(9) Studera nedanstående program:

```
main() {
  int p1[2], p2[2], m=1,n=2,o=3; pipe(p1); pipe(p2);
  if(!fork()) {
    if(!fork()) {
      close(p1[1]); close(p2[0]);
      while(read(p1[0],&n,sizeof(int)))
        write(p2[1],&n,sizeof(int));
      close(p2[0]); close(p1[1]); //* 1
      exit(0);
    }
    close(p1[0]); close(p1[1]); close(p2[1]);
    while(read(p2[0],&o,sizeof(int)))
      printf("%d\n",o);
    close(p2[0]); //* 2
    wait(0); exit(0);
  }
  write(p1[1],&m,sizeof(int)); write(p1[1],&n,sizeof(int)); write(p1[1],&o,sizeof(int));
  close(p1[0]); close(p1[1]); //* 3
  close(p2[0]); close(p2[1]); //* 3
  wait(0); return 0;
}
```

(a) Programmet fungerar som det ska, beskriv programmets funktion och rita ett tidsdiagram som illustrerar de ingående processernas relationer och rita även in de två piparna. Förklara vad som händer när programmet kör och ange dess utmatning.

(b) Vissa rader är markerade med * följt av ett nummer, 1, 2 och 3. På dessa rader finns anrop till **close** som är av central betydelse i IPC (*InterProcess Communication*) - man måste ha koll på sina **close**, annars fungerar det inte. Frågan är: kan några **close** hoppas över? I så fall vilka? Med ”hoppa över” betyder att vi kommenterar bort de rader som är markerade med ett visst nummer, frågan är alltså, kan vi hoppa över (kommentera bort) raden med markerad med 1? Den med 2? De med 3 (vi behandlar dem tillsammans)? Varför, varför inte?

**(6p)**

(10) Studera nedanstående program:

```
int counter=10;
void* tf (void * p){
  pthread_t t = 0;
  while(counter>=0){
    counter--;
    sleep(1);
    if(t==0)
      pthread_create(&t,NULL,&tf,NULL);
  }
  if(t!=0)
    pthread_join(t,NULL);
  pthread_exit(0);
}
main(){
  pthread_t t;
  pthread_create(&t,NULL,&tf,NULL);
  while(counter>=0){printf("%d\n", counter); sleep(1);}
  pthread_join(t,NULL);
}
```

Hur många sekunder kommer programmet att köra? Vad blir utskriften av programmet? Motivera dina svar.

**(4p)**