Tentamen i Operativsystem, HI1025:TEN1 - 8 juni, 2017

Allmänna instruktioner. Tentamen innehåller 9 frågor/uppgifter med totalt 50 poäng. För lägsta godkända betyg (E) krävs ungefär 28 poäng. För att få komplettera (Fx) krävs ungefär 26 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: undantag för uppgift 8.** Av layoutskäl saknas nödvändiga `#include`-direktiv i källkoden i vissa av uppgifterna, men uppgifterna ska behandlas som om direktiven fanns där.

(1) I avsnitt 36 i *Operating Systems, Three Easy Pieces* beskrivs hur olika enheter (*devices*) kan kommunicera med ett operativsystem. De två alternativen *Polling* (också kallat *Programmed IO*) respektive avbrottsbaserad IO (*interrupts*) behandlas. Beskriv i detalj vad dessa respektive sätt innebär **(4p)** och ange exempel på situationer då respektive teknik förekommer **(2p)**. Ange särskilt ett exempel på en situation där *polling* kan vara snabbare än avbrottsbaserad IO. Motivera varför. **(1p)**

(2) För att kunna beskriva och organisera exekvering av maskinkod på en dator under ett operativsystem används termerna *process* och *tråd*. Vidare finns också de fyra termerna *programräknare*, *processbild*, *fildeskriptorer* och *cpuregister*. Förklara var och en av dessa 4 termer och ange, för varje term, huruvida de är gemensamma för alla trådar inom en process eller om varje tråd har en enskild egen instans av det som termen beskriver. **(8p)**.

(3) Förklara vad följande systemanrop gör och vad som kan göra att de misslyckas: a) `fork()` **(2p)**, b) `write()` **(2p)**. c) Varför kan ett *UNIX*-system inte fungera utan `fork()`? **(1p)**.

(4) Förklara följande termer: a) *Round Robin Scheduling* **(2p)**, b) *Mutual Exclusion* **(2p)**, c) *Symbolisk länk* **(2p)**.

(5) Förklara begreppet *context switch*. Förklaringen ska innefatta både context switch mellan processer och trådar **(2p)** och förklara vad som menas med att ”context switch mellan processer är dyrare än context switch mellan trådar” **(1p)** och förklara även varför det är så **(1p)**.

(6) Vad är ett *sidfel* (*page fault*) **(1p)** och vad händer med den process som får ett sidfel? **(1p)**. Hur återhämtar sig en process från ett sidfel? **(1p)**.

(7) Vad menas med att man *partitionerar* en hårddisk? **(1p)**. Vad är en *partition*? **(1p)**. Vad menas med att man *formaterar* en partition? **(1p)**. Vilka av dessa aktiviteter (partitionering/formatering) är nödvändiga om man vill installera ett *UNIX*-system med swappartition på en tom (helt blank) hårddisk? Motiveringar krävs för samtliga aktiviteter som du bedömer nödvändiga **(2p)**.

(8) Experimentell tentauppgift: rätt och fel om trådar och deadlock **(6p)**.
Följande uppgift innehåller 6 påståenden om trådar och deadlock som antingen är riktiga eller felaktiga. Kryssa i vad du bedömer gäller. Rätt svar ger 1 poäng, fel svar ger 0 poäng. För att det inte ska löna sig att chansa är gränsen för godkänt höjd med 3 poäng så om du inte vet någonting, chansa!

1. Om det är möjligt är det alltid bättre att förlägga parallella aktiviteter i parallella trådar snarare än i parallella processer.
   ☐ Rätt ☐ Fel

2. För att deadlock ska kunna uppstå med p-trådar krävs alltid att det finns fler än en resurs/mutex.
   ☐ Rätt ☐ Fel

3. Anropet av `pthread_mutex_lock()` resulterar i ett anrop av en speciell processorinstruktion med läsning och skrivning i en och samma busscykel.
   ☐ Rätt ☐ Fel

4. Deadlock är ett begrepp som berör parallella trådar, vi behöver inte vara oroliga för att deadlock ska uppstå mellan parallella processer.
   ☐ Rätt ☐ Fel

5. Det går inte att starta en process utan att minst en tråd startar.
   ☐ Rätt ☐ Fel

6. P-trådar kan ha individuella lagringsutrymmen genom att lokala variabler deklareras i trådfunktionen, men andra trådar kan få tillgång till dessa lokala variabler genom pekare till de lokala variablerna. Sådana pekare kan till exempel vara deklarerade som globala variabler.
   ☐ Rätt ☐ Fel

(9) Studera nedanstående två program:

```
main() { // a.c kompileras till a
  int p1[2], p2[2], m=1; pipe(p1); pipe(p2);
  if(fork()) {
    dup(p1[0]); close(0);
    close(p1[0]); close(p1[1]);
    dup(p2[1]); close(1);
    close(p2[0]); close(p2[1]);
    execlp("./b","./b",NULL);
  }
  write(p1[1],&m,sizeof(int));
  read(p2[0],&m,sizeof(int));
  wait(0);
  printf("M: %d.\n", m);
}
```

```
main() { // b.c Kompileras till b
  int a;
  while(read(0,&a,sizeof(int))){
    a=a*2; write(1,&a,sizeof(int));
  }
}
```

Två program med flera processer i sig. Vi vill att dessa program ska samverka via `execlp` och vi ser anropet till `execlp` i programmet som heter `a` (till vänster). Som vanligt vill vi att IPC ska karakteriseras av att vi inväntar alla processer och att allting stängs i god ordning. Men när man kör programmet `a` så hänger sig programmet, vi får inte tillbaka prompten då vi gör anropet `./a` i ett kommandoskal ... det finns flera problem med detta program. Vilka är problemen? Förklara vad problemen består i och gör rättelser/kompletteringar av koden så att det fungerar som det är tänkt. Ändra bara i `a.c`, INTE i `b.c` **(3p)**. Ange hur körningen kommer att se ut efter rättelserna **(1p)**. Till sist, rita ett fullständigt tidsdiagram av det rättade programmets körning. I tidsdiagrammet ska också `p1` och `p2` synas **(2p)**. *Att returvärden från systemanrop inte kontrolleras uppfattas inte som ett fel som ska rättas.*