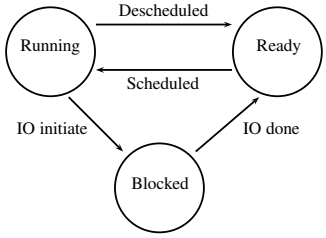
Tentamen i Operativsystem, HI1025:TEN1 - 15 mars, 2017

Allmänna instruktioner. Tentamen innehåller 10 frågor/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.

Uppgifter

(1) Till höger återges figur 4.2 från *Operating Systems, Three Easy Pieces*. Den är ett tillståndsdiagram som visar de olika tillstånden som en process kan anta under sin existens. Förklara vad det betyder för en process att befinna sig i de tre olika tillstånden. Förklara också vad de fyra olika övergångarna innebär och ge konkreta exempel (för varje övergång) på när de kan inträffa **(7p)**.



(2) Återigen i *Operating Systems, Three Easy Pieces* beskriver författarna tre mål med virtuellt minne, de benämner målen *efficiency*, *transparancy* respektive *protection*. Välj två av dessa mål föklara dem och förklara hur ett operativsystem fungerar för att uppnå dessa två mål som du valt **(4p)**.

(3) Förklara följande systemanrop: vad de gör, varför de är nödvändiga och vad som ibland gör att de misslyckas. Dina förklaringar måste använda begreppet *fildeskriptortabell*. Förklara också, mha fildeskriptortabell, hur anropen används tillsammans för att skapa omdirigering **(1p)**. a) `close()` **(3p)**, b) `dup()` **(3p)**.

(4) Vad menas med en *processbild* (*process image*) och hur påverkas den vid anrop till `fork()` respektive `execlp()` **(3p)**? Hur påverkas processid hos en process vid anrop till `fork()` respektive `execlp()` **(1p)**?

(5) En grupp meterologer har tagit fram ett nytt program som skapar väderprogrognoser som de kör på sin superdator som råkar vara ett *UNIX*-system. De kallar programmet `b` och det anropas så här:

```
$ ./b wd
```

där `wd` är en fil som innehåller mätningar från väderstationer som ska processas av `b` som senare levererar en fil som resultat.

Nu vill de att du ska göra ett grafiskt användargränssnitt till systemet och de vill ha en grafisk animation som anger hur långt `b` har kommit och de säger att `b` skickar till sin föräldraprocess upplysningar om hur långt gånget beräkningsarbetet är via en pipe. Du inser att det behövs tre parallella aktiviteter för att klara detta: 1. huvudprocessen som pratar med användaren. 2. Själva beräkningsprogrammet (`b`) och 3. en grafisk animation.

Rita in dessa tre aktiviteter i ett tidsdiagram **(1p)** som anger hur de relaterar till varandra. Ange om de olika aktiviteterna skapats som egna parallella processer eller bara egna trådar och motivera valet av ”egen parallell process / bara egen tråd”. **(3p)**

(6) Ange de tre nödvändiga villkoren för deadlock och ange två sätt att hantera problemet med deadlock som inriktar sig på två av dessa tre nödvändiga villkor för deadlock **(5p)**.

(7) Antag att du har ett system där filen `/etc/fstab` har en rad som börjar så här: ”`/dev/sdb1 /var ...`”. Antag att du som `root` gör följande manövrar vid en prompt: 1. `mkdir /var2`, 2. `mv /var/* /var2`, 3. `rmdir /var`, 4. `mv /var2 /var`. Vad är det tänkt ska ske här? Kommer det att fungera? Om inte vad behövs för att få det att fungera? **(3p)**

(8) Förklara varför parallella trådar hörande till samma process delar sina globala variabler **(1p)** och i förklaringen, se till att det blir tydligt varför motsvarande inte gäller barnprocesser till samma föräldraprocess **(1p)**. Förklara också hur man kan skapa individuella (privata) lagringsutrymmen för parallella trådar inom samma process **(1p)** och förklara varför detta fungerar **(1p)**.

(9) Studera nedanstående kod:

```
void * t(void * p) { // ... code related to thread parameters omitted
  while(read(control_pipe[0],&task,sizeof(int))) {
    pthread_mutex_lock(&m);
    work_with_resource_protected_by_m();
  }
  pthread_exit(0);
}
main() { // Main program assigns tasks to threads via control pipes }
```

Den kod som ges ovan är inte ett fungerande program utan utgör principiella drag i ett program som diskuteras i den här uppgiften. En huvudtråd (`main()-funktionen`) kontrollerar trådar genom att skriva uppdrag kodade som heltal till trådarna genom en pipe som varje tråd inom sin trådfunktion refererar till som `control_pipe`. Vi antar att programmeringsmässiga parametertekniska manövrar med type casting och deklaration av lämpliga parameterstructar finns för att få detta att fungera, dessa manövrar och detaljer utelämnas men uppgiften ska behandlas som om de fanns där. Huvudprogrammet är tänkt att se till att en tråd avslutas genom att skriva heltalet 0 till pipen som då leder till att trådens loop avslutas som leder till att tråden gör en `pthread_exit()` och som leder till att tråden kan inväntas med `pthread_join` i huvudtråden. Varje tråd, då den skapas, får en egen pipe för mottagande av uppdrag från huvudtråden. Det finns dock två problem med ovanstående kod/principdesign. Finn dessa problem, beskriv det problematiska och rätta till det och motivera dina rättelser **(4p)**.

(10) Studera nedanstående två program:

```
main() { // This program is named a
  int p[2], m=1, n=2, o=3; pipe(p);
  if(!fork()) {
    if(!fork()) {
      close(0); dup(p[0]);
      close(p[0]);
      execlp("./b","./b",NULL);
    }
    close(p[0]); wait(0); exit(0);
  }
  close(p[0]);
  write(p[1],&m,4);
  write(p[1],&n,4); // 1
  write(p[1],&o,4); close(p[1]);
  sleep(1); // 2
  wait(0);
}
```

```
// This program is
// named b and is called
// by a.

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

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 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 ... vad är problemet? Förklara vad problemet består i **(1p)** och gör en rättelse/komplettering av koden **(1p)** och ange hur den nya körningen kommer att se ut **(1p)**. Till sist, rita ett fullständigt tidsdiagram över körningen av det rättade programmet **(3p)**, rita in i tidsdiagrammet tidpunkterna som är markerade med kommentarerna `//1` och `//2` i koden till programmet `a` **(1p)**.