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

(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)**

**Svar:** Avbrottsbaserad IO innebär att systemet väntar med att hantera data tills det finns tillgängligt, vid en läsning från till exempel en hårddisk så finns inte data tillgängligt hela tiden pågrund av att hårddisken som levererar data är långsam. Då är det inte meningsfullt att OS stå och väntar utan då installeras en så kallad avbrottrutin som möjliggör att OS i väntan på att data kommer in kan sysselsätta sig med andra saker - annan kod körs, när så data kommer in från enheten aktiveras avbrotssrutinen och OS kan tillgodogöra sig den data som just kommit in. Detta sparar kraft i och med att OS kan vara produktivt om inte data finns tillgängligt. Motsatsen är programmerad IO där OS med jämna mellanrum kontrollerar om data kommit in, det innebär att datorkraft går åt hela tiden kolla om data är tillgängligt, det slösar datorkraft om data väldigt ofta inte är tillgängligt då en kontrolls sker. Avbrottsbaserad IO sker typisk med en hårddisk som nämnt ovan och polling kan användas då vi till exempel har ett snabbt nätverk så att data tas emot genom att OS hela tiden kollar om något nytt kommit in. Detta kan då också vara snabbare än avbrottsbaserad IO efter sådan IO kräver avbrottrutiner och att administrationen kring avbrottshanteringen kan vara en belastning om data kommer in tillräckligt snabbt - nätverken blir ju snabbare och snabbare.

(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)**.

**Svar:** *Programräknare:* fysiskt ett av CPU:ns register som håller reda på vilken instruktion som exekveras, har individuellt innehåll för varje tråd, sköts via context switch. *Processbild:* den del av datorns primärminne som reserveras för en körande process, gemensam för alla trådar inom den processen. *Fildeskriptorer:* en fildeskriptor är ett heltal som anger en plats i processens fildeskriptortabell som kopplar heltalet till någon form av komunikationsinstrument (pipe, socket, etc) eller öppen fil på sekundärt lagringmedium. Gemensamma för alla trådar inom samma process. *CPU-register:* Vid exekvering av maskinkod är dessa register temporära lagringsplatser inuti datorns processor, mycket snabba och med individuellt innehåll för varje körande tråd (sköts via context switch.

(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)**.

a) `fork()`: Skapar en barnprocess som är en kopia av den anropande föräldern. Kan misslyckas om maximala antalet processer redan skapats. b) `write()`: Tar en fildeskriptor som inparameter och skriver till det fildeskriptorn pekar på det som anges i en buffer som också tas som inparameter. Kan misslyckas om fildeskriptorn är stängd eller om användaren inte har skrivrättigheter på destinationen. c) *UNIX*-arkitekturen är fullständigt beroende av `fork()` eftersom det är enda sättet att skapa processer på bortsett från `init` som skapas som första process, alla andra processer har `init` som urförälder.

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

**Svar:** a) Cyklisk schemaläggning där processer får återkommande turer på CPU:n, möjliggör illusionen av samtidigt körande processer när de egentligen turas om väldigt snabbt. b) Ett sätt att turas om

om en resurs så att alltid bara högst en aktör kan hantera resursen åt gången. Poäng har också givits om *intrumentet mutex* har beskrivits. c) En symlink är en fil vars enda innehåll är sökvägen till en annan fil, då man opererar på symlinken sker operationerna på det som symlinken pekar på, fungerar som genväg i Windows fast bättre, (Windows klarar inte genvägar till kataloger).

(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)**.

**Svar:** *Context Switch* kallas CS, hör till concurrency (samtidig exekvering) och är således relevant för både parallellt körande processer och parallellt körande trådar inuti en process. Då det gäller processer turas processerna om att innehaprocessorn och CS är själva bytet då CPU:n överlämnas från en process till nästa. Alla detaljer som hör till körning av en måste då sparas undan till nästa gång processen ska köra och de detaljer som hör till nästa process körning behöver laddas in, saker som är aktuella är programräknarens position, CPU-registrenas innehåll med mera. Det är tid som är sjlva kostnaden. Ett liknande byte behövs då trådar turas om om den tid som CPU:n kör den process de är del av, det som gör en CS för processer vsentligt dyrare (tar mer tid) är att en CS mellan processer är *mycket* mer omfattande eftersom den också behöver byta ut sidtabellen som lagras i MMU:n. Sidtabellen byts inte ut då olika trådar inom samma process körs.

(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)**.

**Svar:** Ett sidfel innebär att en körande process begärt tillgå till en sida som för tillfället inte finns i datorns primärminne. För att processen ska kunna fortsätta köra måste sidan laddas in. Då processen får ett sidfel sövs den och sidan hämtas in till primärminnet, då sidan är inhämtad väcks processen och får fortsätta exekveringen.

(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)**.

**Svar:** Att partitionera en hårddisk innebär att dela upp den mängd spår som den har i delmängder av konsekutiva spår. En partition är just en mängd av på varandra följande spår. Att formatera en partition innebär att införa ett filsystem på partitionen. (Detta raderar all data på partitionen men det viktiga är att filsystemet införs.) Båda aktiviteterna, partionering och formatering är nödvändiga om man ska från en blank hårddisk installera ett *UNIX*-system med swappartition, partitioneringen är till för att överhuvudtaget införa uppdelningen av hårddiskspåren och sedan formateringen inför själva filsystemen på de olika partitionerna. (Swap på den ena och kanske ext4 på den andra.)

(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

   **Svar:** "Alltid bättre" är alldeles för kategoriskt så det är fel. Ett fall då det kan vara bättre att lägga parallella aktiviteter i helt egna processer är då dessa aktiviteter är benägna att krascha. Då drabbas bara en process, inte aktiviteter förlagda i parallella trådar.

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

   **Svar:** Fel, för om samma tråd låser samma mutex två gånger blir det också en deadlock.

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

   **Svar:** Precis så är det. Instruktionen kallas `TAS` vilket står för *Test And Set*. Utan denna obrutna busscykel kan inte *mutual exclusion* ske.

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

   **Svar:** Jodå, deadlock handlar om parallella skeenden, om dessa skeenden sedan förläggs i parallella trådar eller helt egna processer spelar ingen roll. Dealock är aktuellt i båda fallen. Alltså fel.

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

   **Svar:** Rätt, varje process har minst en tråd som kör. Annars skulle ingenting kunna hända i processen.

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

   **Svar:** Det går alldeles utmärkt att göra så.

(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.*

**Svar:** Det finns tre grundläggande fel med program `a.c`. 1. Det ska stå `if(!fork())`, annars så körs inte koden efter i en barnprocess. 2. Omdirigeringarna är felaktiga, det ska vara `close(0); dup(p1[0]);` och motsvarande för `p2`, allstå `close`-anropet först. Slutligen måste vi ha stängningar av piparnas läs och skrivändar i huvudprogrammet så att vi lägger till `close(p1[0]);`, `close(p1[1]);`, `close(p2[0]);` och `close(p2[1]);` innan `wait(0);` i `a.c`. Då kommer programmet att fungera. Utskriften blir `M: 2.` (följt av ny rad) och följande är ett tidsdiagram över relationen mellan de ingående

processerna (de tjocka pilarna representerar piparna, den till vänster är `p1` och den till höger är `p2`):