Operativsystem ID2200/06

omtentamen

2017-04-10 14:00-18:00

Namn: _____

Instruktioner

- Du får, förutom skrivmateriel, endast ha med dig en egenhändigt handskriven A4 med anteckningar. Mobiler etc skall lämnas till tentamensvakterna.
- Svaren skall lämnas på dessa sidor, använd det utrymme som finns under varje uppgift för att skriva ner ditt svar.
- Svar skall skrivas på svenska eller engelska.
- Du skall lämna in hela denna tentamen.
- Inga ytterligare sidor skall lämnas in.

Versioner

Denna tentamen gäller för flera olika omgångar av kurserna ID2200/06. Beroende på vilken kursomgång du följer så skall olika delar av tentamensfrågorna besvaras.

För omtentander i ID2200 gäller följande:

- Registrerade för tentamen på 6hp, VT16 och HT16: besvara frågorna 1-9, inte fråga 10.
- Registrerade för tentamen på 3.8 hp, dvs före VT16: besvara frågorna 1-8, inte 9-10.
- För de som är registrearde för tentamen på 3.8hp men som ännu inte har lab-momentet avklarat kan man besvara även fråga 9 och då få det momentet tillgodoräknat. Fråga 9 hanteras separat så få poäng på fråga 9 kompensears inte av flera poäng i övriga delar.

För omtentander i ID2206 gäller följande:

- Registrerade för tentamen på 6hp, HT16: besvara frågorna 1-9, inte 10.

- Registrerade för tentamen på 4.5hp, före HT16: besvara frårna 1-8 och fråga 10
- För de som är registrearde för tentamen på 4.5hp men som ännu inte har lab-momentet avklarat kan man besvara även fråga 9 och då få det momentet delvis tillgodoräknat. Fråga 9 hanteras separat så få poäng på fråga 9 kompenseras inte av flera poäng i övriga delar.

Betyg för 6hp

Tentamen har ett antal uppgifter där några är lite svårare än andra. De svårare uppgifterna är markerade med en stjärna, *poäng**, och ger poäng för de högre betygen. Vi delar alltså upp tentamen i grundpoäng och högre poäng. Se först och främst till att klara grundpoängen innan du ger dig i kast med de högre poängen.

Notera att det av de 40 grundpoängen räknas bara som högst 36 och, att högre poäng inte kompenserar för avsaknad av grundpoäng. Gränserna för betyg är som följer:

- Fx: 21 grundpoäng
- E: 23 grundpoäng
- D: 28 grundpoäng
- C: 32 grundpoäng
- B: 36 grundpoäng och 12 högre poäng
- A: 36 grundpoäng och 18 högre poäng

Gränserna kan komma att justeras nedåt men inte uppåt.

Gränsen för E är för tentamen på 4.5hp 18 poäng och för 3.8hp tentamen 16 poäng.

Erhållna poäng

Skriv inte här, detta är för rättningen.

Uppgift	1	2	3	4	5	6	7	8	9
Max G/H	4/0	2/2	4/2	4/2	4/2	4/2	4/2	2/2	12/8
G/H									

Totalt antal poäng:

Namn:_____ Persnr:_____

1 Operativsystem

1.1 vad händer här? [2 poäng]

Om vi ger komandona nedan, efter varandra, i ett *shell*; vad kommer resultat att vara?

```
> mkdir foo
> cd foo
> echo "hello hello" > tomat.txt
> mkdir ../bar
> ln tomat.txt ../bar/gurka.txt
> rm tomat.txt
> cd ../
> wc -w bar/gurka.txt
```

Svar: Resultatet blir att `2` skrivs ut eftersom vi räknar antal ord i filen `gurka.txt`.

1.2 kommandon i ett shell [2 poäng]

Ge en kort beskrivning av vad kommandona nedan gör.

- `cat`
- `less`
- `ln`
- `mv`

Svar: Slå upp deras betydelse med hjälp av `man`.

Namn:_____ Persnr:_____

2 Processer

2.1 vad är var? [2 poäng]

I koden nedan har vi allokerat upp två arrayer; vilka arrayer och i vilka segment återfinns de: globalt, stack eller heap?

```
#include <stdio.h>
#include <stdlib.h>

int x = 43;

int h[] = {1,2,3,4};

int *foo(int *a, int s) {

  int *r = malloc(s * sizeof(int));

  for(int i = 0; i < s; i++) {
     r[i] = a[i] + x;
  }
  return r;
}

int main() {

  int *c = foo(h, 4);

  printf("%d \n", c[3]);

  return 0;
}
```

Svar: De två arrayerna är `h` som allokeras globalt och `r` som allokeras på heapen i `foo()`.

Namn:_____ Persnr:_____

2.2 privilegierade instruktioner [2 poäng*]

I en x86-arkitektur så är vissa instruktioner priviligerade och endast möjliga att utföra i *Ring-0*. Som exempel kan vi ta: HLT (halt), LIDT (load interupt descriptor table), MOV CR (skriver till Control Register). Hur använder vi denna egenskap för att kunna implementera ett operativsystem på ett effektivt sätt?

Svar: Genom att dessa instruktioner kan vi låta användarprocesser köra i s.k. *limited direct execution*. Användarprocesserna kör i *Ring-3* och kan göra allt utom att ändra på de delar som operativsystemet skall hantera. Utan detta stöd från hårdvaran skulle vi inte våga låta en användarprocess få göra någonting direkt utan allt skulle vara tvunget att gå via operativsystemet.

3 Schemaläggning

3.1 shortest time-to-completion first [2 poäng]

Schemaläggaren "shortest time-to-completion first" är en optimal schemaläggare; vad är det den optimerar och varför är den i praktiken inte användbar?

Svar: Den optimerar omloppstiden (*turnaround*) för ett givet antal jobb. Problemet är att vi i de flesta fall inte vet hur lång tid det kommer att ta för ett jobb att avslutas. Att vi kanske inte får en så bra reaktionstid eller att långa jobb kommer att ta väldigt långt på sig om de hela tiden blir avbrutna av kortare jobb är i sig inte ett problem eftersom det är omploppstiden som skall optimeras.

3.2 reaktionstiden [2 poäng]

När vi vill minska reaktionstiden så kan vill vi helst kunna avbryta jobba även om de inte är klara. Om vi gör detta så har vi en parameter som vi kan välja, genom att anpassa denna så kan vi förbättra reaktionstiden. Vad är det för parameter som vi kan sätta? Hur skall vi förändra den och vilka oönskade konsekvenser kan det få?

Svar: Vi kan minska på den tid som varje jobb tillåts köra innan vi byter till nästa jobb. Detta medför att jobb som är klara att köra mycket snabbt blir schemalagda. Nackdelen är att vi förlänger den genomsnittliga omloppstiden och i värsta fall spenderar en stor del av tiden på att växla mellan processer.

Namn:_____ Persnr:_____

3.3 realtidsschemaläggare [2 poäng*]

I realtidsschemaläggning så beskriver vi jobb med tre värden som brukar betecknas: e, d och p. Vad står dessa parametrar för (du behöver inte komma ihåg vilken som är vilken men skall kunna ange egenskaperna som beskriver ett jobb).

Svar:

- e: längsta exekveringstid
- d: dead-line, när måste jobbet vara klart
- p: periodicitet, hur ofta skall jobbet schemaläggas

4 Virtuellt minne

4.1 paging [2 poäng]

Antag att vi har en virtuell adress som består av ett 22-bitars sidnummer och en 12-bitars offset. Vi har en fysisk adressrymd på 32 bitar, kodar ett ramnummer i 20 bitar och element i en sidomvandlingstabell är fyra bytes stort. Hur stor är en sida och hur stor skulle en komplett omvandlingstabell behöva vara?

Svar: Sidorna är 4Kibyte stora eftersom vi har 12 bitars offset. I uppgiften står det att sidnummret kodas i 22 bitar vilket skulle innebära 4Mi sidor. Om varje sida skall representeras av 4 byte i en tabell så har vi en tabell på hel 16Mibyte.

4.2 omvänd sidtabell [2 poäng]

Om vi har ett fysiskt minne som är betydligt mindre än de virtuella minnen som operativsystemet erbjuder kan det vara idé att implementera en så kallad omvänd sidtabell. Hur fungerar en omvänd sidtabell och vad ger den för fördel?

Svar: En inverterad tabell har en rad per fysisk ram. I raden beskrivs vilken process och vilken sida som finns i ramen. Vid om vandling måste en process söka igenom hela tabellen (implementeras i hårdvar eller med hjälp av en hashtabell) för att hitta vilket ramnummer som gäller. Fördelen är att alla processer använder samma tabell och att tabellens storlek är proportionell mot antalet ramar inte mot det virtuella minnet.

Namn:_____ Persnr:_____

4.3 bounds [2 poäng*]

När vi implementerar ett segmenterat minne så vill vi gärna ha ett *bounds-värde* som säger hur stort segmentet är. Om vi inte har det så riskera vi att adressera utanför segmentet. När vi implementerar ett sidat minne (*paging*) så behövs inget bounds-värde? Varför behöver vi inte det? Vad är det som hindrar oss från att adressera utanför sidan?

Om det nu är något som hindrar oss från att göra fel, är det något som kostar? Vad kostar det?

Svar: Alla sidor är lika stora och storleken är exakt så stor som vårt offset kan addressera dvs vårt offset kommer alltid att hamna innaför sidans gränser. Kostnaden är att det vi alltid allokerar minne i storleken av en sida. Om vi har en sidstorlek på 4K byte och behöver 6K byte, så kommer vi att allokera 8K byte dvs vi får en viss intern fragmentering.

5 Minneshantering

5.1 tältsemester [2 poäng]

När jag rest runt och tältat på olika campingplatser i Sverige (det har jag inte) så har jag märkt att campingplatser har helt olika system för var man skall slå upp sitt tält. En del säger att man skall slå upp det i närheten av de andra, men se till att man lämnar ett lucka på fyra meter till nästa tält. Andra platser har rutat in ett område i lika stor a rutor och säger att jag kan slå upp mitt tält i ruta 17.

Vad är problemet med de två olika strategierna och vad har detta med operativsystem att göra?

Svar: I det ena fallet får vi extern fragmentering när mindre tält flyttar och luckorna är för små för att utnyttja till större tält. I det andra fallet har vi ett viss spill för intern fragmentering. Detta är samma problem som vi har för minnesallokering i ett operativsystem.

5.2 statisk reallokering [2 poäng]

Istället för att implementera dynamisk minneshantering så kan man göra en statisk reallokering av program när de läggs in i minnet. Vad innebär statisk reallokering, vad är det som måste göras när ett program läggs in i minnet? Ge ett exempel.

Namn:_____ Persnr:_____

Svar: Vid statiskt reallokering så ändras ett programs alla minnesoperationer så att de arbetar mot den position som programmet faktiskt har allokerats. En `MOV`-instruktion som i koden läser från position `0x200` skulle om programmet läggs in på minnesposition från `0x5000` ändras till att läsa från `0x5200`.

5.3 kod, data, stack... [2 poäng*]

Vid implementering av segmenterat minne är en lösning att till exempel låta de översta bitarna i en virtuell adress avgöra vilket segment som skall användas. En annan lösning är att helt enkelt ha separata segment för kod, data och stack. Om vi skulle ha dessa tre segment, hur skulle då processorn avgöra vilket segment som skall användas när vi gör en operation mot minnet?

Svar: De olika segmenten adresseras med olika register: *instruktionspekaren* refererar till koden, *stackpekaren* och *baspekaren* refererar till stacken och alla övriga indirekta eller direkta referenser är till datasegmentet.

Namn:_____ Persnr:_____

6 Flertrådad programmering

6.1 count [2 poäng]

Vad kommer skrivas ut om vi exekverar proceduren `hello()` nedan samtidigt i två trådar? Motivera svaret.

```
int loop = 10;

void *hello() {
  int count = 0;

  for(int i = 0; i < loop; i++) {
      count++;
  }
  printf("the count is %d\n", count);
}
```

Svar: Varje tråd har sin version av `count` och de två trådarna kommer alltså inte att störa varandra. Varje tråd kommer därför att skriva ut the count is 10".

6.2 men varför [2 poäng]

Om vi har en multicore-cpu så är det klart en fördel att arbeta med flera trådar eftersom vi då kan utnyttja beräkningskraften bättre. Om vi nu sitter på en maskin med enbart en kärna så är det väl rätt meningslöst att dela upp sitt program i flera trådar, eller? Kan ett program som är uppdelat i trådar exekvera snabbare även om vi bara har en kärna att köra på? Motivera.

Svar: Ja, om vi har ett program som gör I/O kan en en tråd vara suspenderad i väntan på I/O medan en annan tråd fortsätter med beräkningen.

6.3 gröna trådar [2 poäng*]

En del operativsystem erbjuder s.k. *gröna trådar* dvs trådar som är implementerade med hjälp av biblioteksfunktioner och som hanteras av användarprocessen. Nämn en fördel och en nackdel med gröna trådar.

Svar: En fördel är att man kan byta mellan trådar betydligt snabbare eftersom man inte behöver göra något systemanrop. Nackdelen är att om användarprocessen blir suspenderad av operativsystemet så stannar alla trådar,

vi kan till exempel inte utnyttja flera kärnor eller fortsätta exekvering i en annan tråd om vi gört en I/O-operation.

7 Filsystem och lagring

7.1 en helt vanlig HDD [2 poäng]

Om en hårddisk har en medelsöktid (*average seek time*) på 10 ms, en rotationshastighet på 7200 rpm (*rounds per minute*) och har en läshastighet på 200 MiB/s. Vad är då medeltiden för att läsa en slumpvald sektor på 4 KiB?

Svar: Ca: 14 ms. Om det tar 10 ms att positionera armen, och i snitt 4 ms (halva tiden av 60/7200 ms att rotera ett var) så kommer vi att hitta sektorn på 14 ms. Att läsa en sektor på 4KiB är i sammanhanget försumbart, när vi väl har huvudet plats så tar själva läsningen bar 20 mikrosekunder.

Namn:_____ Persnr:_____

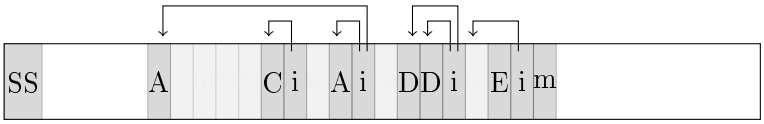
7.2 sektorer till filsystem [2 poäng]

En hårddisk består av ett antal sektorer. Beskriv en enkel implementation av ett filsystem och hur filsystemets datastrukturer kan lagras på en hårddisk.

Svar: Ett enkelt filsystem skulle kunna ha en ett *suber-block* i första sektorn som beskriver filsystemets övergripande struktur: var bitmappar, inoder och datablock finns. Bitmapparna innehåller information om vilka inoder och datablock som är lediga. Filer och mappar är representerade av en inod var. Inoden innehåller meta-information och pekare till datablock som har innehållet i filen eller mappen.

7.3 loggbaserat filsystem [2 poäng*]

Nedan ser du en schematisk bild av ett loggbaserat filsystem. Om systemet nu börjar få ont om plats så kommer det att försöka skapa plats. Hur kan mer plats skapas och hur kommer systemet att se ut efter det att mer utrymme tillgängliggjorts?



Svar: Mer plats skapas genom att filer som ockuperar block i systemet "svans", kopieras till dess huvud. I exemplet ovan kan man vias att filen `A` har kopierats och därmed frigör konsekutivt minne fram till blocket som används för filen `C`.

Namn:_____ Persnr:_____

8 Virtualisering

8.1 varför [2 poäng]

Beskriv ett viktigt skäl till varför man vill köra flera virtualiserade operativsystem istället för ett enda.

Svar: Ett skäl är att man har tillämpningra som kräver olika versioner av ett operativsystem eller helt olika system. Istället för att låta dessa köra på egna underbelastade maskiner kan man låte dem samsas på en maskin.

8.2 kernel/user mode [2 poäng*]

Antag att vi har en hypervisor som kör i *kernel mode* och det virtualiserade systemet kör i *user mode* för att skydda hypervisorn. Varför kan vi inte låta det virtualiserade operativsystemet ligga kvar i *user space* när en av dess processer kör?

Svar: Om det virtualiserade operativsystemet är kvar i *user space* så är det inte skyddat från den körande processen.

Namn:_____ Persnr:_____

9 Implementering

9.1 stacken [2 poäng]

Nedan ser vi ett program som skriver ut innehållet på stacken.

```
void zot(unsigned long *stop, int a1, int a2, int a3, int a4, int a5, int a6) {
  unsigned long r = 0x456;
  unsigned long *i;
  for(i = &r;  i <= stop; i++){
    printf("%p      0x%lx\n", i, *i);
  }
}

int main() {
  unsigned long p = 0x123;

  zot(&p,1,2,3,4,5,6);
 back:
  printf("  back: %p \n", &&back);
  return 0;
}
```

Vid en körning får vi följande utskrift. Beskriv de värden som pekas ut med pilar (<--).

```
0x7ffeb3331f58      0x456
0x7ffeb3331f60      0x7ffeb3331f60  <-- ??
0x7ffeb3331f68      0x3a7dbfad7df4b100
0x7ffeb3331f70      0x7ffeb3331fa0
0x7ffeb3331f78      0x400663    <-- ??
0x7ffeb3331f80      0x6   <-- ??
0x7ffeb3331f88      0x4004a0
0x7ffeb3331f90      0x123
  back: 0x400667
```

Svar: Den översta är värdet på variabeln `i`, som ligger på stacken. Den pekar på sig själv just när den skrivs ut. Den andra är en instruktionspekare som är den instruktion som vi skall fortsätta med när vi är klara med `zot()`, den pekar som som synes till en posistion 4 bytes innan back. Den sista är den sexa som vi ger som argument när vi anropar `zot()`. De övriga argumenten ligger i register.

Namn:_____ Persnr:_____

9.2 sbrk() [2 poäng]

Proceduren `malloc()` är en biblioteksrutiner som använder sig av systemanropet `sbrk()` för att allokera minne. Vad är fördelen för en process om den använder `malloc()` istället för att anropa `sbrk()` direkt?

Svar: När den använder malloc så undviker den att göra ett dyrt systemanrop och kan mycket enklare lämna tillbaks minne m.h.a. `free()`. Det återlämnade minnet kan sedan återanvändas utan något systemanrop.

9.3 pipes [2 poäng]

Om vi har två processer, en producent och en konsument, som kommunicerar via en s.k. *pipe*. Hur kan vi då förhindra att producenten skickar mer information än vad konsumenten kan ta emot och därmed får systemet att krascha?

Svar: *Pipes* har en inbyggd flödesontroll, om konsumenten inte väljer att läsa kommer producenten att bli suspenderad när den försöker skriva.

9.4 tmpfs [2 poäng]

Vad gör kommandot nedan och varför skulle vi vilja göra det? Vad är nackdelen?

```
> sudo mount -t tmpfs tmpfs ./tmp
```

Svar: Kommandot monterar ett temporärt filsystem under mappen `tmp`. Filsystem finns bara i RAM vilket gör att det har bättre läs- och skrivprestanda. Nackdelen är att alla filer försvinner när vi avslutar operativsystemet.

9.5 byt kontext [2 poäng]

Programmet nedan leker med sitt eget s.k. *kontext*. Vad innehåller strukturen `ucontext_t` och vad blir resultatet när vi kör programmet?

```
#include <stdlib.h>
#include <stdio.h>
#include <ucontext.h>

int main() {

  int done = 0;
```

```
  ucontext_t one;
  ucontext_t two;

  getcontext(&one);

  printf("hello %d\n", done);

  if(!done) {
    done = 1;
    swapcontext(&two, &one);
  }
  return 0;
}
```

Svar: Strukturen innehåller en sparad kopia på processens register, däribland kod- och stackpekare. Om `done` ligger på stacken så får vi utskriften "hello 0","hello 1". Om `done` ligger i ett register (kompilerat med `-O2`) så kommer vi ligga i en oändlig loop eftersom variabeln alltid får tillbaks sitt ursprungliga värde.

9.6 läshastighet [2 poäng]

Hur stor är skillnaden i läshastighet om vi jämför att läsa från från minnet med att läsa från en fil på en roterande hårddisk (första läsningen av filen)?

- $10ns$ vs $1\mu s$
- $10ns$ vs $10ms$
- $100ns$ vs $10\mu s$
- $1\mu s$ vs $1ms$

Svar: Att läsa från minnet tar i storleksordningen några nanosekunder. Att läsa från en fil på disk kan ta upp emot 10 millisekunder.

9.7 ett huvud och en fotnot [2 poäng*]

Vid implementation av minnesallokering så är det vanligt att man ha ett gömt *huvud* placerat alldeles innan den minnesarea som man delar ut. I detta huvud kan man bland annat skriva hur stor arean är så att det blir enklare att ta hand om arean när vi gör *free*. Man kan även använda sig av en gömd fotnot som ligger efter arean där man kan skriva att arean är använd eller inte och kanske en pekare till huvudet. Vad är det för poäng med att lägga den informationen efter arean, räcker det inte med huvudet?

Svar: När vi gör free på en area kan vi titta i fotnoten på den area som ligger alldeles innan den area som vi vill göra free på. Om den arean är fri så kan vi kanske slå ihop blocken till ett större block. Vi kan som vanligt titta i arean framför oss, där är det ingen skillnad.

9.8 lite bättre [2 poäng*]

Så kallade *pipes* är ett mycket enkelt sätt att skicka data från en process till en annan. Det har dock sina begränsningar och ett bättre sätt är att använda s.k. *sockets*. Om vi istället för en pipe öppnar en *stream socket* mellan två processer så har vi flera fördelar. Beskriv två saker som en *stream socket* ger oss som vi inte får om vi använder en *pipe*.

Svar: En *stream socket* ger oss en dubbelriktad kanal där vi kan välja flera olika addresseringsmetoder. Om vi använder *pipes* är vi begränsade till filsystemet för addressering, om vi använder sockets kan vi addressera en process på en annan dator.

Namn:_____ Persnr:_____

9.9 character device [2 poäng*]

Vi kan skapa en s.k. *character device* och interagera med den med hjälp av `ioctl`. I koden nedan, beskriv vad `fd`, `JOSHUA_GET_QUOTE` och `buffer` är och hur vårt *device* kan tänkas fungera.

```
if (ioctl(fd, JOSHUA_GET_QUOTE, &buffer) == -1) {
  perror("Hmm, not so good");
} else {
  printf("Quote - %s\n", buffer);
}
```

Svar: Parametern `fd` är en fildeskriptor som vi får när vi öppnar den fil som modulen (*devicet*) har registrerat sig på. Parametern `JOSHUA_GET_QUOTE` är en kodad intruktion som beskriver vad vi vill ha gjort, om vi har givit en buffer och om den skall användas för läsning eller skrivnig. Den tredje parametern `buffer` är en minnesarea där modulen kan läsa från eller skriva till. I detta exempel kan man tänka sig att vi begär ett citat från modulen och denne skriver citatet i den angivna buffern.

9.10 delat minne [2 poäng*]

Processer i ett Unix-system kan kommunicera med varandra via flera olika kanaler. De kan även dela minnesareor på liknande sätt som två trådar i en process can dela *heap* och global data. Hur kan vi få två processer att dela minne?

Svar: Vi kan använda systemanropet `mmap()` som mappar en fil i det virtuella minnet. Om två processer mappar samma fil och anger att den är delad mellan processer är det den ene skriver direkt synligt för den andre. Vi kan även ange att en process skall dela minnesutrymme med din moderprocess om vi använder `clone()` istället för `fork()`.

Namn:_____ Persnr:_____

10 Bara för omtentamen i ID2206 reggade före HT16

10.1 publika-nyklar [2 poäng]

Beskriv vad man kan göra med s.k. assymetriska krypteringsmetoder får vi har en publik och en provat nyckel som vi inte kan göra med system där vi bara har en nyckel.

Svar: Vi kan skapa digitala signaturer och autentisera någon.

10.2 hårda och mjuka [2 poäng]

Vad är skillnaden mellan vad som informellt kallas "hårda" och "mjuka" realtidssystem?

Svar: I så kallade hårda system så har vi tidsgränser som vi garanterat måste hålla. I ett mjuk system har vi gränser som är mer eller mindre viktiga, vi kan bryta dem men vi måste notifiera användaren på något sätt så att man kan hantera situationen.

10.3 multiprocessor [2 poäng*]

Om vi har en multiprocessor så måste en schemaläggare naturligtvis kunna låta processer köra på de olika processornerna. Vi kan dock göra bättre eller sämre schemalägging, beskriv en aspekt som vi måste ta hansyn till och hur vi anpassar schemaläggaren.

Svar: Vi måste till exempel ta hänsyn till vilken kärna en process körde på senast och i möjligaste mån låta processen köra på samma kärna får att förbättra cache-prestanda. Schemaläggaren kan ha en kö per kärna och endast flytta processer mellan kärnor om balansen blir alltför snedfördelad.

10.4 distribuerat operativsystem [2 poäng*]

Man kan låta ett operativsystem spänna över flera datorer s.k. distribuerat operativsystem istället för att ha oberoende system på de enskilda datorerna. Vad gör det för skillnad när vi skall implementera ett system med kommunicerande processer.

Svar: Vi kan till exempel låta processer kommunicer genom delat minne ellr *pipes*. Dessa former av kommunikation är normalt begränsade till en

maskin men det distribuerade systemet skall ge oss den möjligheten oavsett var någonstans en process råkar köra.