Kapitel 4 (DeBlanche) **– Datorarkitektur**

Faktiskt är mycket av det som står i detta kapitel mer på sin plats i parallellkursen i mikrodatorteknik. Men kurserna och ämnena går förstås i varandra. Skissen på sid 31 kommer ni att veta mycket väl vad den innebär under mikrodatorteknikkursen.

4.1 Processorer

Processorn är datorns centrala resurs, det är den absolut viktigaste resursen som det är operativsystemets uppgift att fördela. Alla uppgifter som datorsystemet har kräver förstås processorkraft. En processor är egentligen en automat, ni har realiserat automater i digitalteknikskursen och vet att en automat har ett antal tillstånd och ett antal in- och utsignaler. En CPU är alltså en automat, men med ett oerhört stort antal tillstånd.

4.2 Vad gör en processor

En processor kör instruktioner som en programmerare har radat upp, *C*-program kompileras till körbara program i maskinkod och det är denna maskinkod som processorn kör (exekverar).

Avsnitten 4.3-4.7 beskriver sedan de olika delarna i en datorarkitektur. Läs om detta. Det är operativsystemets uppgift att administrera alla dessa saker. Det är här viktigt att minnas att operativsystemet själv är också ett stort program som ligger i datorns minne. (Det betyder alltså att OS måste så att säga vakta sig själv.)

Kapitel 5 – Vad gör ett operativsystem

Kort svar: OS administrerar datorsystemets alla resurser så att de uppgifter som datorsystemet ska utföra så att uppgifterna blir utförda på bästa sätt.

Läs kapitel 5 för att få en överblick av vad detta betyder. Vi kommer också efter genomgången kurs förstå mer av vad OS gör.

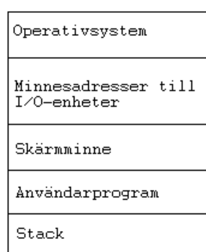
Kapitel 6 – Klasser av operativsystem

6.1 Batchsystem

Batchsystem var den första typen av enkla OS som bara skickade in ett program i datorn, dåförtiden lagrades program på hålkort eller magnetband och datorn körde programmet och levererade utdata också på hålkort eller magnetband eller liknande. Det är förstås förlegat idag, men de stora superdatorerna som har enorm beräkningskapacitet, kan ibland sägas operera enligt denna princip. Det är inte ovanligt att en väderleksprognos (som kräver enorma mängder beräkningar) körs på en superdator under det att inget annat sker. Så principen batch-processing finns fortfarande idag. *Batch* är engelska för "*grupp*" eller "*bunt*" och refererar troligen till att man hade en bunt hålkort som man laddade i datorn. Batch-processing idag betyder "En sak i taget och avbryt inte denna sak."

6.2 Enanvändarsystem

Nästa steg på skalan är enanvändarsystem. *Single-task operating system*. Här görs fortfarande bara en sak i taget, men den sak som görs kan avbrytas. Datorns minne organiserades då någonting i stil med nedanstående figur



En del för operativsystemet, en del för I/O-enheter (som då var åtkomliga via primärminnet), en del för skärmminnet (idag lagrar ofta grafikkorten det som ska finnas på skärmen), en del för användarprogrammen och en del för stacken.

Vi har här också glidit över lite i historia, det här är så som datorer fungerade förr i tiden. Det är inte en historiekurs vi går, men vi kan ändå, genom att studera historian förstå hur dagens arkitektur fungerar.

Det grundläggande sättet för ett single-task OS är alltså: ladda in programmet som ska köras i areans för användarprogram och starta exekveringen där. Då kör en process. Ordet "task" betyder på engelska uppgift och är här synonymt med process. Istället för single-task skulle vi lika gärna kunna säga single-process och den enda process som kan köra är alltså det program som för närvarande är laddat.

6.3 Fleranvändarsystem (Multitasking OS)

I ett multi-tasking-, multi-user-OS måste alla saker finnas som finns i ett single-task OS, och mycket mer. Nu ska flera processer (alltså program, tasks) kunna köra samtidigt och alla dessa processer kan inte ha egna enheter, alla processer måste dela på de befintliga resurserna. Hårdvaran delas alltså och den administration som behövs för att sköta delandet utförs av operativsystemet.

Då vi har flera användare innebär det att de måste skiljas från varandra. Det innebär att de måste styrka sin identitet – lösenord behövs. Detta är anledningen till att inloggning införs.
Normalt finns en användare som har alla rättigheter, denna användare kallas ibland *super-user* eller systemadministratör. I *Linux/UNIX* har han/hon användarnamnet **`root`**.
Man talar ofta om privilegier som användare har. Vanliga användare har inte så många privilegier. Däremot har `root` alla privilegier som finns. Vissa kommandon är *priviligierade*.

Two-mode operation: Körning kan ske i två olika tillstånd. När en användarprocess körs så ställs datorn in för användarkörning. När sedan ett avbrott ges (OS tar över), eller ett systemanrop görs så kan tillståndet behöva gå över i priviligierat tillstånd. Det här kommer vi att studera djupare senare.

I/O & minnesskydd: användarprocesser får bara ett visst minnessegment att använda. Olika processer har olika segment i minnet och får inte rota i varandras kod eller data. Hårdvara ser till att detta efterlevs. Om man, trots allt, lyckas rota i varandras data/kod så genereras ett segmenteringsfel. Detta kallades ”allmänt skyddsfel” i *Windows* förut. I allmänhet innebär det att

den process som försökte göra något otillåtet avslutas. I *UNIX*-system kallas det *segmenteringsfel* "Segmentation Fault". Det uppkommer till exempel om man till exempel i *C* skriver `scanf("%d", i);` och glömmer adressoperatorn på `i` som här då är en heltalsvariabel.

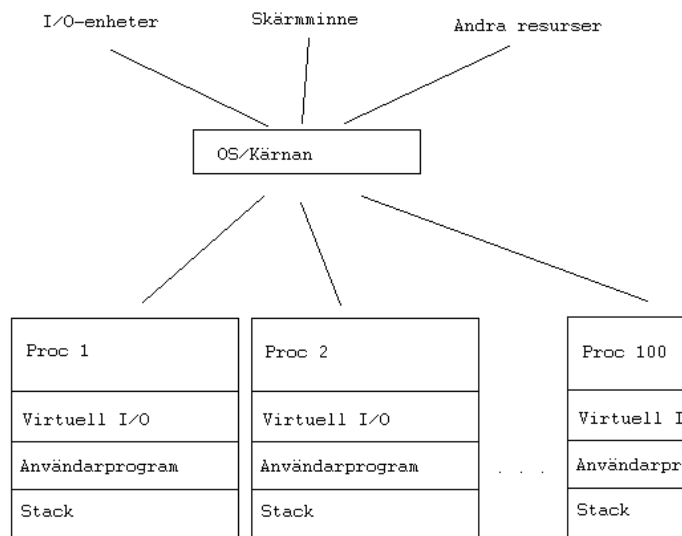
Time-Sharing/tidsdelning

Det finns väsentligen två sätt att dela på CPUn:

1. OS kör användarprogrammen. Då släpps aldring CPUn helt och fullt. Instruktioner kollas innan de körs och detta är snarare emulering än exekvering. Det är dyrt eftersom det blir så mycket overhead.

2. OS kontrollerar en hårdvarumässig timer (typ äggklocka) som sätts och sedan släpper OS CPUn helt till en körande process. Efter en stund ger timern ett avbrott och processen får lämna ifrån sig CPUn till OS som troligen då lämnar vidare den till nästa process i kön. (Round-Robin). Nu kör varje process helt och fullt utan detaljkollas av OS. Det är effektivt och således inte så dyrt. Overhead har minskat väsentligt jämfört med förra metoden. Det är den andra metoden som används mest.

Vi ser på en principiell minneskarta i ett multitasking-, multiuser-OS:

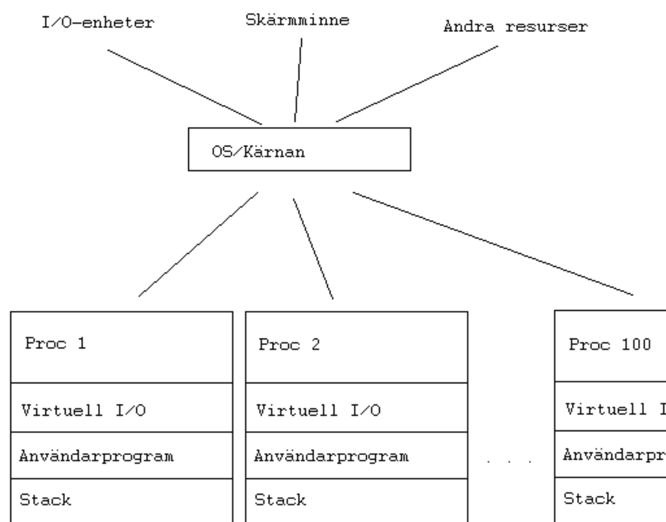


Varje körande process som vill komma åt I/O-enheter eller andra delade resurser måste gå via OS genom ett så kallat *systemanrop*. OS hanterar sedan det hela så att varje process upplever att den har egen tillgång till de enheter den anropar, därav namnet "Virtuell I/O" i rutorna ovan. Vidare kan ett minnesarrangemang ju inte se ut som ovan, datorns minne är ju en lång rad av utrymmen, så ovanstående skiss är endast en principiell skiss. Vi kommer att se tydligare skisser som går djupare in i detaljerna på hur det egentligen ser ut senare.

6.4 Distribuerade operativsystem och 6.5 Realtidsoperativsystem: Läs själva

Grundläggande om processer

Den viktigaste poängen att få här ifrån är förståelsen för att ett OS i allmänhet kör olika processer och att alla dessa processer delar på CPU-tiden (*time-sharing*, via *round-robin*) och även på utrymmet i datorns primärminne (*minneshantering* via *virtuellt minne*). Ingen process har alltså (givetvis) ensamrätt på resurser i datorsystemet. Operativsystemet fördelar resurserna mellan alla processer och vi har tidigare sett följande skiss, som vi nu ska förtydliga:



De två teknikerna som är viktiga att förstå här är alltså *round-robin* och *virtuellt minne*. (Senare ska vi även se på sekundärminneshantering, partitioner och filsystem och sånt.)

Processer

En process är en aktivitet i datorn som kör, alla program som kör kräver processer för att köra. De enklaste programmen behöver bara en process medan med komplicerade program som tex en webbserver, kan bestå av flera samverkande processer. Att köra flera processer parallellt innebär då att datorsystemet kör flera parallella aktiviteter. Här ska vi se närmare på hur detta fungerar. Senare kommer begreppet *tråd* och att nämnas, vi behöver inte riktigt veta vad det är vid det här tillfället, vi kan låtsas att det bara är ett annat ord för process (vilket är absolut fel i klassisk mening) och konsistent förstå den teori som presenteras. Så vi låtsas det en stund så kan vi fördjupa oss i vad trådar egentligen är mer noggrannt senare. Vi ska nu gå igenom lite olika aspekter av processer för att förtydliga begreppet ytterligare.

Skapande av processer

Det finns tre sätt att skapa en process säger författaren till boken *Operativsystem*, genom att starta operativsystemet, genom att en process startar en annan process eller genom att användaren startar en process "interaktivt". Men om vi ser noggrannare på dess tre olika sätt så ser vi att sista sättet och andra sättet har det gemensamt att "en process startar en annan process" eftersom ju användaren kör ett program som *består av processer* och "användaren startar en process" innebär ju att användaren instruerar sitt program, *som består av körande processer*, att starta en process, så det andra sättet kan egentligen också hänföras till kategorin att det är en *process som startar en process*. Då finns första fallet kvar, att en process startas genom att operativsystemet startar och här startas faktiskt en process på ett annat sätt än via en process. Så man kan argumentera för att processer endast kan uppkomma på två sätt (inte tre),

1. Genom start av operativsystemet
2. Genom att en process startar en annan process.

I ett *UNIX/Linux*-system är det så här: för att starta en process måste en annan process anropa operativsystemet och begära att en annan process startas (systemanrop till kärnan). Problemet är då hur ska den första processen startas? Svaret blir att den startas då operativsystemet startar och operativsystemet startar endast en gång och denna process blir den enda enda process som startar som *inte* behöver startas av en annan process. Det är processen `init`. Varje process har ett unikt processidentifikationsnummer mellan 1 och 65535 och `init` har processid 1. Processen `init` är alltså den enda process som inte är skapad av en annan process, alla andra processer har en skapare som är en process.

Processhierarki

Varje process skapas alltså av operativsystemkärnan genom ett så kallat *systemanrop*. Den process som begär att en ny process ska skapas brukar som sagt ibland oegentligt kallas *processens skapare* det är egentligen OS kärna som skapar processen i respons till ett systemanrop från en process. Den process som utför systemanropet om att en ny process ska skapas brukar mer ofta kallas *förälder*, eller *föräldraprocess*, (*parent process* eller bara *parent*). Den process som skapas kallas då förstås en *barnprocess* eller *barn,* (*child process* eller bara *child*.) Med den här terminologin kan man säga att processen `init` (i ett *UNIX/Linux*-system) är den enda processen som inte har någon förälder. Eftersom en barnprocess är en annan process än sin förälder har de olika processidentifikationsnummer. Varje process kan förstås bli förälder till flera barnprocesser och en hierarkisk struktur uppstår men processen `init` högst upp som urförälder till alla processer i hela systemet. Samtliga barn hörande till en och samma föräldraprocess kallas en *processgrupp*.

Processtillstånd

En process kan vara i något av följande tillstånd:

1. *New*. Ny, alltså nyss skapad, ännu inte redo att köra.
2. *Ready*. Redo att köra, den är lagd i en kö och beredd att få tillgång till en processor/processorkärna.
3. *Running*. Processen kör, det vill säga processens maskinkod exekveras av en processor/processorkärna.
4. *Sleeping*. Processen sover, det vill säga väntar på att få köra igen eller väntar på I/O. Detta kallas också för att processen "blockerar".
5. *Terminated*. Processen har kört klart och är avslutad men har inte hunnit laddas ur systemet ännu.

Den allmänna gången är ofta att en process skapas som barnprocess med ett speciellt syfte, ja, alla processer, utom en, `init`, är ju barnprocesser. När syftet är uppfyllt och barnprocessen blir avslutad, *Terminated*, ska föräldraprocessen normalt läsa av status. Operativsystemet väntar med att ta bort administrativa datastrukturer till dess föräldern läst av status. Men om inte föräldern kan läsa av status, tex om föräldern själv avslutas eller är i en oändlig loop, ja då kommer aldrig det avslutade barnets status att läsas av. En process som befinner sig det här tillståendet, alltså *Terminated* - men aldrig uppföljd, kallas för en *Zombie*, och tillståndet kallas *Defunct*. Den är då en levande död som aldrig går i graven och tas bort helt ur systemet. Det är potentiellt skadligt eftersom operativsystemet slösar administrativa resurser på detta sätt. När vi programmerar måste vi

absolut undvika att skapa zombier. (Det finns mekanismer för att överlåta föräldraskapet på andra processer om en förälder till barn skulle dö.)

Processavslutning

En process kan avslutas på två principiellt olika sätt, dels genom att den kör klart sin kod och väntar på att bli uppföljd av sin förälder eller så kan processen avslutas utifrån. När en process kör så säger man att den lever och när den inte kör är den död så att avluta en process är synonymt med att döda processen. I *UNIX/Linux*-världen sker det genom att man skickar signalen `KILL` till processen som då direkt avslutas. Om en process inte dödas utan får köra klart så kan den göra det på två principiellt olika sätt: den kan lyckas eller misslyckas. I *UNIX/Linux*-världen lämnar den avslutade processen efter sig en så kallad *exit-kod*, koden 0 betyder lyckades, koder skilda från noll indikerar olika problem.

Elementär processkommunikation – *IPC* via signaler

I *UNIX/Linux*-världen kan processer kommunicera med varandra på många sätt, vi ska programmera mycket med *IPC* (*Inter-Process Communication*) senare i kursen. Men det finns ett basalt eller elementärt kommunikationssätt som vi kan illustrera redan nu, det kallas *signaler*. Då en process dödas skickar man signalen `KILL` till den, den signalen har nr 9 (*UNIX/Linux*) men många av de grundläggande manövrarna som görs indikeras till processerna genom signaler. Några exempel:

9 – `KILL` – processen som får signalen 9 ska omedelbart ovillkorligen avslutas
15 – `TERM` – processen som får signalen 15 får en begäran om att avsluta sig själv. Processen kan alltså då i lugn och ro spara sina data och sedan avsluta sig själv.
11 – `SEGV` – segmenteringsfel, processen har försökt referera till minne den inte få röra, det är detta som händer vid `scanf("%d",i);` om `scanf` behöver att variabeln `i` har `&` på sig.
17 – `CHLD` – en barnprocess är avlutad.

Det finns många fler än så här, vi kommer troligen inte att programmera så mycket med dem, men ni ska veta att de finns. Man skickar signaler till en process genom att använda kommandot `kill`. Observera att kommandot heter `kill` men används till att skicka alla signaler, att själva kommandot heter `kill` får nog tolkas som att det var det enda kommandot gjorde från början, dödade (avslutade) processer. Läs manualsidan till `kill`.

Time-sharing – tidsdelning – schemaläggning på en dator med EN processorkärna

Författaren till kurslitteraturen skriver "... en dator [kan] bara göra en sak åt gången... ". Det är oklart vad författaren menar, troligtvis menar författaren att en datorn med bara en processorkärna kan endast köra en process åt gången. Men det är inte riktigt sant heller, det finns någonting som heter DMA-kanaler (som ni bör läsa om på *Wikipedia*) så hela formuleringen "... en dator [kan] bara göra en sak åt gången... " är tyvärr ganska olycklig. Vi ska formulera oss på ett annat sätt, vi ska tala i mer precisa termer kring schemaläggning (*scheduling*) på ett datorsystem som har endast EN processorkärna. Det betyder att systemet vi betraktar endast har en processor.

Antag alltså att vi har ett system med en processor som endast har en processorkärna. Här är den precisa formuleringen av den olyckliga formuleringen ovan: "Endast en process kan vara i

tillståndet *running* åt gången." Processer kan som sagt ha olika tillstånd, *running*, *sleeping*, *terminated* etc. När en process är *running* (körande) så är den i besittning av en processor/processorkärna och eftersom det system vi nu betraktar har endast en processorkärna så kan endast en process åt gången vara i tillståndet *running*. Men kruxet är att vi kör ju system som faktiskt verkar göra flera saker på samma gång! Flera program kan ju köra även om vi bara har en processorkärna, hur kan det komma sig? Det är här som begreppet *time-sharing*, *tidsdelning*, kommer in. Operativsystemet låter de processer som behöver CPU:n turas om att köra CPU-tiden delas ut. För att möjliggöra detta måste ett schema skapas och det måste införas en policy för hur man tar bort CPU:n från en process. Olika approcher finns där:

* *Strictly non-preemptive:* Varje process fortsätter att inneha CPUn tills processen är klar eller behöver I/O (eller något annat inträffar.) Detta var sättet i *Windows 95* eller *Macintosh System 7*.

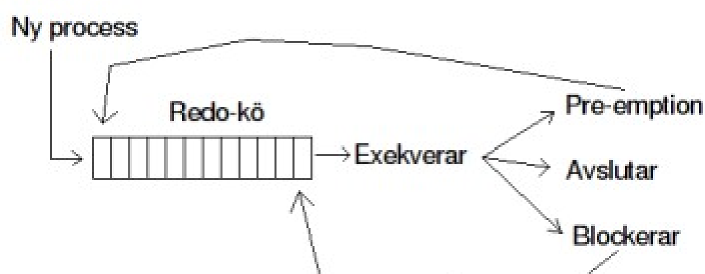
* *Strictly preemptive:* Systemet avbryter en process obönhörligen då dess time-slice är slut. Det vill säga, då äggklockan ringer så är det tack och adjö, punkt slut, nu ska nästa process ha CPUn.

* *Politely-preemptive:* Systemet avbryter en process då dessa time-slice är slut, men om processen är inne i en kritisk sektion (ett avsnitt som bara MÅSTE fullbordas) får processen göra klart den. Det är så här som *Windows* och *UNIX/Linux* fungerar och som de flesta väljer att fungera. Vi ska senare studera mer vad som menas med ”*kritisk sektion*”.
Vad vill vi uppnå med scheduling?

1. Maximal användning av datorsystemet.
2. Så många jobb ska bli klara så fort som möjligt.
3. För interaktiva processer: kort responstid. Ett interaktivt program måste svara snabbt på en användarinteraktion, till exempel musklick, knapptryckningar etc.

Dessa tre krav måste man ibland kompromissa mellan. Det finns olika typer av uppgifter som bör hanteras på olika sätt av samma system. Därför har man inför så kallade *scheduling hierarkier*/*nivåer*. Med olika nivåer så kan olika policyer tillämpas inom varje nivå. Till exempel så kan vi ha *Queueing* för batch-job (ofta utskrifter som ju inte ska vara samtidiga, det går ju inte att tidsdela en skrivare) och *Round robin* för interaktiva processer (ofta applikationer som körs av en användare).

Det här är en skiss som jag ritat av från boken.

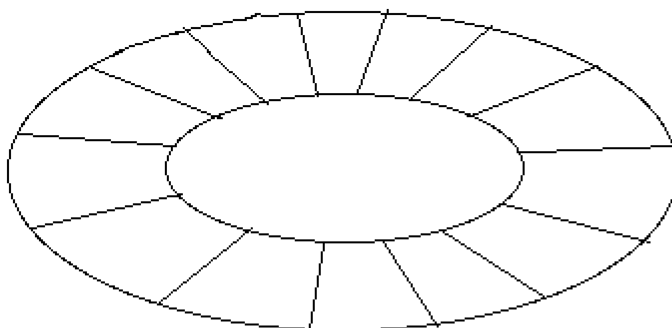


Skissen beskriver rent alllmänt hur en process rör sig genom de olika tillstånden. Lägg märke till hur en process efter den har "blockerat", alltså släppt CPU:n frivilligt, kanske för att använda en resurs, hamnar först i kön då den är klar med resursen. Detta kan till exempel vara en hårddiskaccess eller att en nätverksanslutning lyckats. Detta händer inte efter pre-emption, efter pre-emption har ju en process använt klart hela sitt tidskvanta på CPU:n . Vi kan diskutera mer här.

Olika schemaläggningstekniker

Att schemalägga processer innebär att skapa en tilldelning av operativsystemet resurser till de processer som ska köra. Det viktigaste resursen är CPU:n och därför brukar schemaläggning handla mycket om hur CPU:n schemaläggs. (Andra resurser behöver dock också schemaläggas.)

Round-robin: Varje process tilldelas ett fixt tidskvanta och sedan tilldelas CPU:n dessa processer i ett cykliskt förlopp, en process exekverar i säg 100 mikrosekunder och få sedan ställa sig sist i kön för att få en ny 100 mikrosekunders time-slice. Vi kan ovan i skissen se hur en process är i kön, blir körande, blir pre-empted (berövad CPU:n) och hamnar sist i kön och så går det hela vidare. Ur CPU:ns perspektiv kan det se ut så här:



Mycket enkel skiss som bara illustrerar tiden och att CPU:n kommer tillbaka till samma process igen. Den cykliska strukturen ska representera tiden som CPU:n spenderar på varje process. Man kan ge processer olika prioritet på olika sätt, dels kan man förlänga eller förkorta den tid som en process får köra på CPU:n och/eller så kan man ge en process tillgång till CPU:n mer eller mindre ofta.

FIFO: First-In-First-Out, är en schemaläggningsteknik som inte är cyklisk, en process (eller ett *jobb* som det då kallas) schemaläggs och kör tills den är klar. Exempel på detta kan vara utskriftsjobb eller stora beräkningsjobb som kör på superdatorer. Vi kommer inte att grubbla så mycket över denna typ av schemaläggning.

Lottoschemaläggning: En rättvis form av schemaläggning där CPU-tid lottas ut. I längden jämnar det ut sig och blir som *round robin*. Författaren hävdar att denna schemaläggningsteknik är fri från låsningar och det förstår jag inte alls. Vi kommer att återkomma till denna fråga när vi studerar låsning. Vi kommer dock inte heller att grubbla så mycket över denna typ av schemaläggning heller. Prioritet kan förstås tilldelas en process här på samma sätt som vid round robin genom att man till exempel förlänger det tidsintervall som en process får inneha CPU:n.
Trådar, alltså avsnitt 8.8 och kapitel 9, *Concurrency och låsningar*, kommer vi att vänta med till slutet av kursen och då kommer vi att studera dessa ämnen med annan litteratur.

Ovan nämnde vi att vi här har beskrivit schemaläggning på en dator med EN processorkärna, jag har inte studerat något annat, men en kvalificerad gissning är att schemaläggning på system med flera processorkärnor innebär något liknande, men skillnaden i skissen ovan blir att en redo-kön kan betjänas av flera processorer/processorkärnor.

Kapitel 10 Minneshantering

Vi har ovan sett hur processer delar på *tiden* som resurs, CPU-tiden. Nu ska vi ser hur processerna delar på datorns primärminne, alltså datorns interna *utrymme*.

Allmänt hur en process är organiserad i primärminnet
En process består av två saker: data och instruktioner. **Data** innebär saker som lagras, i respons till att vi till exempel i *C* deklarerar en variabel som heter `a` av typen `int` skapas ett minneutrymme i datorns minnesarea när processen kör som är på 4 bytes. Detta är alltså **data**. Vad är **instruktioner** då? Jo, antag att vi i samma *C*-program deklarerar två andra variabler, `b` och `c`, också av typen `int`, då skapas ytterligare två platser där annan data lagras. Antag nu att vi i vårt *C*-program har satsen `c=a+b;`, i respons till detta kommer det att genereras en maskinkodsintruktion av följande typ:

Skapa ett temporärt lagringsutrymme, kalla det `TMP`.	
Lägg in innehållet på adressen till `a` i `TMP`.	`MOV A,TMP`
Addera, till `TMP`, innehållet på adressen till `b`.	`ADD B,TMP`
Lägg innehållet i `TMP` på adressen till `c`.	`MOV TMP,C`

Det ovanstående är ungefär vad en osmart kompilator skulle göra. En smartare kompilator kanske skulle kunna skapa följande kod istället:

Lägg innehållet på adressen till `a` på på adressen till `c`.	`MOV A,C`
Addera, till innehållet i adressen till `c`, det som finns på adressen till `b`.	`ADD B,C`

Den sista varianten är endast två instruktioner (`MOV`, `ADD`) och den tidigare varianten är tre (`MOV`, `ADD`, `MOV`). Det är inte så viktigt, viktiga här är att illustrera vad maskinkod är, och det är viktigt att här förstå att det som står ovan inte är maskinkod, utan en *skisserad assemblerkod*. Varje assemblerinstruktion (som `MOV` och `ADD`) har dock exakta maskinkodsmotsvarigheter och det är detta som då blir **maskinkodsinstruktioner**. En instruktion har alltså tre delar:

Angivelse av vilken operation som ska utföras (t. ex. `ADD`, `MOV` etc.)
Operand 1 (t. ex. adressen till innehållet i `a`, `b`, `c` eller `TMP`.)
Operand 2 (t. ex. adressen till innehållet i `a`, `b`, `c` eller `TMP`.)

I exemplen ovan har vi två operander (*operand* betyder det som en operation ska utföras på), för det är en väldigt vanlig processorarkitektur. Minns att dessa instruktioner bildar insignaler då till processorn som ju kan ses som en automat som ni studerat i digitalteknikskursen. Vi kan nu tänka oss en minneskarta för en körande process enligt följande:

```
| Data          |
| Data          |
| Instruktioner |
| Instruktioner |
```

Detta kallas en *processbild* (*process image*) och innehållet av en processbild består alltså av processens data och processens instruktioner. Data och instruktioner kan förstås vara blandade, men den viktiga poängen här är att överallt i data och intruktionerna finns referenser till *adresser*, alltså positionsangivelser i minnet. Kruxet är, hur ska dessa positionsangivelser hanteras?

Varför är det ett problem, hur positionsangivelserna ska hanteras, jo, vi vill kunna flytta omkring en processbild (eller delar av den) i datorns minne *när processen kör*. Operativsystemet ska kunna omorganisera hur processers processbilder förläggs under körning. Det är ett väldigt lätt problem om man bara har en process (*Single-task OS*), men ett avancerat problem då man har *flera* processer körandes samtidigt (*Multi-process OS*) särskilt då faktiskt processernas processbilder delas upp i mindre delar via sidhantering (*paging*).

Vi ska inte först studera problemet då en processbild delas upp i mindre delar (sidor/ramar), vi ska börja studera problemet som om en processbild bara var en solid klump av på varandra följande minnespositioner med data och instruktioner som innehåll. Vi ska beskriva en modell som är en starkt förenklad variant av det som egentligen gäller.

För att möjliggöra att en processbild ska kunna flyttas omkring så införs begreppet *basadress*. Alla adressangivelser i processbilden, då den ligger på disk, blir således relativa referenser. När en processbild laddas in i minnet, för att processen ska köras igång, adderas alltså basadressen till samtliga adressangivelser i processbilden då den ligger på disk. Under körning kan alltså en processbild flyttas genom att själva bilden flyttas till de nya adresserna, men alla adressangivelser måste alltså uppdateras för att processen ska kunna fortsätta köra som tänkt. Vi inför här logiska och fysiska adresser. Då processbilden ligger på disk (i form av ett program som kan köras) kan vi tala om att alla adresser är *logiska*, det är så som processen tänker sig körningen. Vi ser på ett par principskisser som illustrerar detta:

Skisserad maskinkod av `c=a+b;`:	Det som faktiskt ligger på disken (i en binärfil)	Minneskartan så som processen uppfattar den:

`MOV A,TMP`
`ADD B,TMP`
`MOV TMP,C`
	`MOV 20,16`
	`ADD 24,16`
	`MOV 16,28`
		`TMP` ligger på plats 16
		`A` ligger på plats 20
		`B` ligger på plats 24
		`C` ligger på plats 28

Positionerna, alltså adresserna, av `TMP`, `A`, `B` och `C` är alltså 16, 20, 24 respektive 28. Dessa adresser är de som processen uppfattar dem och kallas som sagt logiska adresser. När nu dock operativsystemet laddar in processbilden i primärminnet för körning bestäms att programmets basadress ska vara 1024, då adderas detta tal till alla adresser i processens processbild och data och instruktioner läggs in på de ställen som behövs. Den resulterande processbilden i datorns primärminne visas till höger.

Fysiska adress	Innehåll
1040 (=1024+16)	Värdet på TMP
1044 (=1024+20)	Värdet på A
1048 (=1024+24)	Värdet på B
1052 (=1024+28)	Värdet på C
1056 (=1024+32)	`MOV 1044,1040`
1060 (=1024+36)	`ADD 1048,1040`
1064 (=1024+40)	`MOV 1040,1052`

Adresserna 1040-1064, som är de logiska adresserna adderade till basadressen är de adresser som processen faktiskt använder vid konkret körning. Dessa adresser kallas då *fysiska* adresser. Fråga att fundera över: Var är `TMP`, `A`, `B` och `C` i den laddade processbilden? I binärfilen? (Det finns ett litet fel i processbilden som vi kan diskutera, men vi vill främst bara illustrera ett begrepp med den.)

Logiska och fysiska adresser och översättning

Vi har här två viktiga begrepp som kommer att möjliggöra ett av de centrala teknikerna inom operativsystemteknologi, *logiska* och *fysiska adresser*,

Logiska adresser är alltså det som processen uppfattar, en process kan uppfattas som ett program som kör och när det programmet kör har det en uppfattning om hur data och instruktioner är organiserade, det har en uppfattning om "hur världen ser ut". Men, när processen/programmet laddas in i minnet ändras adresserna i processbilden (beroende på vari minnet processbilden hamnar, alltså beroende på basadressen.) De adresser som verkligen innehåller processens innehåll kallas *fysiska adresser*. Det här skisserar en metod som är använd i vissa operativsystem:

1. Då en process ska köras pekas först maskinkodsfilen ut som innehåller processens program.
2. Minne reserveras för processbilden (data och instruktioner) och de binära data som ligger i den körbara filen börjar läsas in.
3. Det program som läser in den binära körbara fil och skapar processbilden i minnet heter laddaren, och laddaren omvandlar då de adresser som finns i filen (logiska adresser) till fysiska adresser.
4. När processbilden är uppbyggd läggs processen i kön för att vara redo att köra.

Nu går det inte riktigt till så här i alla system, en annan, mer dominerande metod (*paging*) ska vi titta på snart, men det viktiga här är att det sker en översättning, från adresser som finns på disk (logiska adresser) till de fysiska adresserna. Laddaren (*loader*) utför denna uppgift. Det som beskrivs ovan är dock en bra modell som vi kan utveckla våra programvara ifrån. Som utvecklare behöver man inte fundera över översättning mellan logiska och fysiska adresser.

Paging – sidhantering

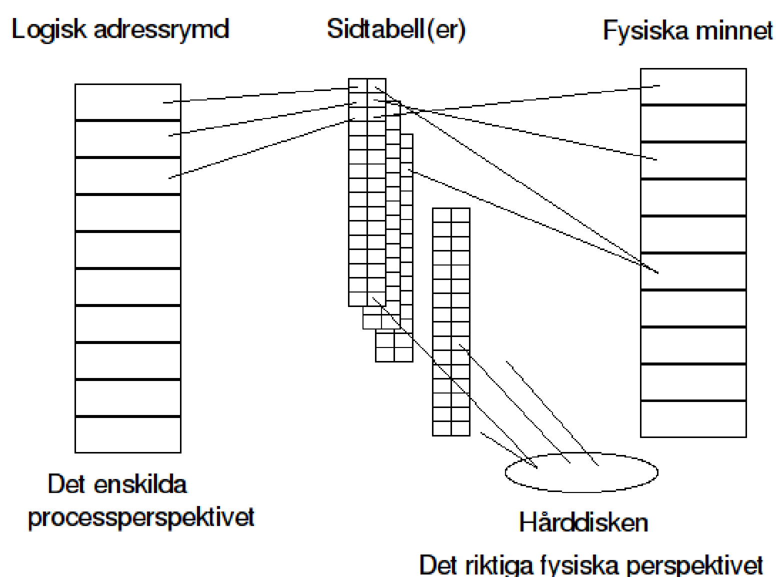
Nu ska vi beskriva ett annat sätt som ger större flexibilitet och som är det sätt som används mest, *Paging*, alltså *sidhantering*.

Grunden i sidhantering

För att möjliggöra sidhantering delas processbilden i den logiska adressrymden för en process upp i ett antal sidor, *pages*. Varje sida innehåller således data och/eller instruktioner för en process som ska köra. En normal sida är inte så stor, i *Linux* ett par kB. Det är här viktigt att sidstorleken är precis samma för alla processer i hela systemet. Det fysiska minnet, å andra sidan, delas då upp i ramar, *frames*, och när en process ska laddas in för körning så väljs ett antal ramar ut där dess sidor ska in. Sedan laddas ett antal av processens sidor in, det behöver inte vara alla sidor som laddas in, vissa sidor kanske innehåller kod som inte används så ofta, då kan de sidorna få vara kvar i processbilden på hårddisken.

Översättningsförfarandet i sidhanteringen

Som vi såg ovan behövs en pågående översättning mellan logiska och fysiska adresser. När vi har en fysisk adressrymd som är sammanhängande (som i fallet med basadress) är denna översättning enkelt att göra, vi adderar basadressen till de logiska adresserna i processens adressrymd och så får vi de fysiska adresserna. Det här fungerar inte då vi har sidhantering för när vi nu sönderdelar en logisk adressrymd i sidor som vi lägger i olika ramar, lite här och var i det fysiska minnet (beroende på var det finns plats) måste översättningen bli mer komplicerad, vi får införa en sidtabell (*page table*) som hela tiden håller reda på i vilken ram en viss sida ligger. Vi ser på en figur för att förtydliga det hela.



Varje enskild process upplever minnet som en följd av sidor som den har tillgång till. Alla sidor finns tillgänliga, "*Random Access*". Vi har "Det enskilda processperspektivet" till vänster. Men, för att innehållet på en sida verkligen ska vara tillgängligt, vare sig det är data som ska läsas eller skrivas eller instruktioner som ska köras så måste innehållet på sidan vara inne i datorns fysiska minne, den måste vara inläst. Översättningen från logisk adress (vilken sida är vi på?) till fysisk adress (i vilken ram ligger sidan?) sker genom att en tabell konsulteras. Detta kallas som sagt *sidtabellen* och observera här då att vissa sidor hörande till en körande process kanske inte finns i minnet! För att processen då ska kunna köra vidare måste denna sida läsas in. Man säger att då inträffar ett *sidfel* (*page fault*.) I figuren ovan är flera sidtabeller, hörande till olika processer, inritade. Vi ser även att vissa ramar i det fysiska minnet innehåller en sida som tillgås av flera processer. Det illustrerar olika möjligheter till besparingar som paging, sidhantering, medför.

MMU

Hur går körningen till? Jo, så fort en process begär tillgång till minnet (vilket händer varje klockcykel eftersom alla instruktioner till en process ligger i minnet) så konsulteras sidtabellen. Vid varje klockcykel konsulteras alltså sidtabellen och översättningen (logisk->fysisk adress) äger rum genom att tabellen anger den ram där sidan är placerad. När detta är klart kan sidans innehåll kommas åt. Själva avläsningen och uppdateringen av sidtabellen (vid sidfel etc) hanteras av en

speciell krets som heter *Memory Management Unit*. Det är en krets som jobbar mycket tätt tillsammans med CPU, förr var det en separat krets, nu är den ofta inbyggd i CPU. MMU innehåller den sidtabell för den process som för närvarande körs. När en annan process kör så måste en annan sidtabell laddas in i MMU.

Virtuellt minne

Med *paging*, sidhantering, blir det fysiska minnets organisation bortkopplat från processernas medvetande. Varje process upplever sig ha konstant tillgång till de olika sidorna i sin processbild, men det är som vi sett inte säkert att en viss sida för närvarande ligger i primärminnet. Denna isärkoppling möjliggör *virtuellt minne*. Operativsystemet kan presentera en möjlighet för processer att köra med ett primärminne som överstiger storleken på det konkreta fysiska minnet! Hurdå, jo, genom att använda diskutrymmet. Det kan alltså mycket väl vara så att att 200 processer kör som upplever att primärminnet är 4GB, att alltså de alla har tillgång till 4GB, medan dator som det här kör på i själva verket endast har 2GB internminne.

Allmän virtualisering

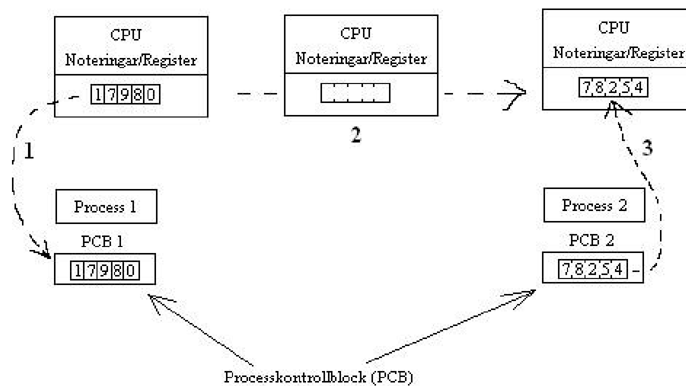
De här begreppen illustrerar också virtualisering i allmänhet. Vi har en klient som vill tillgå en resurs, i det här fallet är klienten en process och resursen är minnet, klienten samexisterar med andra klienter som också vill tillgå resurser av samma slag, i det här fallet har vi andra processer som också behöver tillgå fysiskt minne. Alla klienter (processer) får vända sig till en resursfördelare som erbjuder dem en gränsyta mot resursen, alla klienter kan samexistera genom resursfördelaren och upplever sig ha tillgång till resursen i fråga. Resursfördelaren här är operativsystemet och minnet blir då, som sagt, virtuellt. En process vet inte var det fysiska minnet som den har tillgång till är exakt och då, kan vissa delar av minnet faktiskt finnas på hårddisken i swaparean. Vi kan dra oss till minnes bilden i början på föreläsningen som beskriver detta: ett OS är en stor virutaliseringsprogramvara och klienterna är processer och det som virtualiseras är hela systemets resurser: hårddisk, primärminne, in- och utmatningsenheter, ja allting. Det betyder alltså att begreppet virtualiserin varit med ända från början då de första (riktiga) OS:en (läs *UNIX*) skapades.

Sidvalsalgoritmer

Då ett sidfel inträffar och en sida som inte finns i minnet måste läsas in så måste ofta en annan sida kastas ut. Det är inte bra att köra med för många sidor i minnet. Frågan är vilken sida ska kastas ut? Det finns olika algoritmer. *Least-Recent-Used*, *Random*, *FIFO*, ja, många. Läs om dessa själva.

Context Switch och processkontrollblock

Enligt *round-robin* ska ju varje process köra lite grann och sedan lämna över till en annan process. När en process då lämnar över till en annan så måste alla data hörande till den förra processens körande sparas undan och data hörande till den kommande processen måste läsas in. Detta gäller CPU:ns register, men även filtabeller och den stora sidtabellen.



Ovan ser vi hur en context-switch går till, noteringarna och de register som process 1 hade då den lämnar ifrån sig CPU sparas undan i ett så kallat processkontrollblock (*PCB*, *Process Control Block*) och sedan går CPU över till nästa process. Eftersom detta även händer med hela sidtabellen, MMU laddas med en ny sidtabell, och en tidigare sidtabell sparas undan, så är en context-switch en tidskrävande procedur. Man skulle då kunna tänka sig att det vore bra med så få context-switchar som möjligt, men då kan man å andra sidan köra färre processer.

Thrashing

Ett sidfel leder mycket troligt till en context-switch. Det är så eftersom den process som kör och råkar ut för ett sidfel behöver diskaccess. Det finns då en risk med att köra för många olika processer. Många olika processer begär hela tiden tillgång till många olika sidor. Om tillräckligt många processer begär tillgång till tillräckligt många olika sidor kan följande hända:

1. Process *Arne* får sidfel och vill läsa in sin sida 10. Process *Arne* får då alltså vila en stund medan OS inleder sidfelsprocess och ger hårddisken i uppdrag att läsa in sida 10.
2. En massa andra processer kör och samtidigt väntar process *Arne* på att få börja köra, till slut lyckas OS läsa in process *Arne*s sida nr 10 *Arne* väcks och får köra lite grann.
3. Då kommer process *Bertil* och får också ett sidfel. Det händer sig då som så att process *Bertil*s sida knuffar ut process *Arne*s sida! Det innebär att process *Arne* knappt hinner köra någonting alls innan stackarn får ett nytt sidfel.

Om vi har tillräckligt många olika processer och ett ganska fullt primärminne så kan ovanstående scenario inträffa. Då kan det hända att operativsystemet använder mer tid till att swappa in och ut sidor istället för att få något vettigt arbete gjort, alltså köra processer. Detta kallas då *Thrashing* och innebär att systemet går mycket långsamt.

Tidskomplexitet jämfört med utrymmeskomplexitet

Rent abstrakt kommer ni kanske att kunna observera att det finns en konflikt mellan tid och utrymme, om man minskar på tiden som en process får behöver den mer utrymme för att kunna göra det arbete den ska och tvärtom, om man minskar på utrymmet som en process får behöver den ofta mer tid för att utföra det den ska.

Sekundärminne och *UNIX* modularitet

Termen *RAM*-minne är en förkortning som betyder *Random Access Memory* och förknippas traditionellt med det som kallas datorns *primära minne*. Termen ”*Random Access*” betyder ”tillgång (access) på vilken plats som helst (random)” och just en dators primära minne sitter i minneskretsar och där kan man få tillgång till vilken plats som helst på samma tid vilket inte är fallet med hårddiskar där vi visserligen kan få tillgång till alla platser, men det tar olika lång tid beroende på hur de mekaniska delarna i hårddisken är placerade. Helt enkelt, om läshuvudet är i ena änden av disken och vi behöver läsa från andra änden av disken så måste först läshuvudet flytta sig. En icke-teknisk term för detta är ”ställtid”, traditionella hårddiskar är alltså långsamma på grund av ställtider. Men på senare år har det kommit något som heter SSD = *Solid State Drive*, termen *Solid State* betyder att den inte har några rörliga delar, dessa lagringsmedia är baserade på flashminnen, ungefär som USB-minnen, och har alltså inga ställtider utan kan faktiskt också sägas kunna beskrivas med ”tillgång på vilken plats som helst”, det vill säga *Random Access*.

Termen *RAM*-minne är alltså inte så bra för att definiera termen primärminne. Bättre är följande definitioner:

Primärminne: De minnesceller som töms då datorn stängs av. Går alltid först till primärminnet innan den kan komma åt saker i sekundärminnet, det är därför primärminnet heter just detta, ”först” betyder ”primär” här.

Sekundärminne: De minnesceller som behåller sitt innehåll då datorn stängs av.

Vi kommer här att fokusera på *Sekundärminnet* och hur det hanteras. Vi ska inte göra hårklyverier här, men det är värt att poängtera att vissa termer inom datortekniken är problematiska.

Det kan ofta vara bra att ta det allra allra senaste med en nypa salt eftersom utvecklingen går så fort så kommer just den kunskapen att bli förlegad ganska snart. Vidare är det inte så intressant att lära sig en massa utantillfakta, det är mer intressant för oss att se på de allmänna principerna bakom sekundära lagringsmedia. (Vi avstår från att behandla magnetband även om det är ett viktigt sekundärt lagringsmedium.)

1. Diskar

Vi ska börja rent konkret och tala om den klassiska disken, den har hängt med länge. För att lagra data måste man ha ett medium som kan anta olika tillstånd och olika tillstånd representerar då olika tillstånd. Det mest två utbredda teknikerna för att få en disk att lagra data genom att vara ett medium som antar olika tillstånd är att diskytan görs magnetisk eller fås att reflektera ljus på olika sätt. Här skiljer man mellan magnetiska lagringsmedium (som lagrar data genom att anta tillståndet magnetiserat/omganetiserat) och optiska lagringmedium (som lagrar data genom att reflektera ljus eller inte). Magnetisering och avmagnetisering har visat sig vara en snabb process och lämpar sig då för varierande datamängder, det är därför en dators hårddisk är magnetisk. Optiska lagringsmedia går inte lika fort att förändra varför de helst används i CD och DVD-skivor, här behövs inte lika mycket förändringar. I gengäld är ett optiska lagringsmedium mycket säkrare än ett magnetiskt, det är väldigt lätt att avmagnetisera en hårddisk och därmed förstöra den data som den bär. Det betyder att optiska lagrinsmedia lämpar sig mycket väl för arkivering, säkerhetskopiering och transportering av data. Det är därför som programvara ofta levereras på CD-skivor. (Men det ändras också nu.)

1.1. Fysisk struktur av diskar

En disk är indelad i sektorer, precis som RAM är indelat i ramar (frames). En sektor är den minsta adresserbara enheten på en disk och den är normalt på 512 bytes. Det finns sektorer på hårddiskar och CD-skivor (och med CD-skiva menar vi också DVD och *BlueRay*, de tillhör alla filsystemet ISO-9660), men på CD-skivorna är sektorer inte lika viktiga som på en hårddisk. Varför inte? Jo accesser (läsning eller skrivning) till en CD-skiva sker ofta i ett enda långt svep medan accesser till en hårddisk sker mer hoppigt, hit och dit. Därför är data på en CD organiserad i ett enda långt svep, data ligger som en i ett enda långt spår som är EN *spiral* som börjar inne i centrum på skivan och sträcker sig gradvis utåt. En hårddisk har dock sin början på yttersta delen av diskytan och har FLERA *spår* (eller *cylindrar* som det också kallas) som har högre ordningsnummer ju längre in på skivan man kommer.

1.2. Termen ”Cylinder”

Magnetiska hårdiskar brukar vara en hel stapel med diskar, det betyder att om man betraktar alla spår som ligger på samma avstånd från centrumaxeln som diskarna snurrar runt så bildar denna stapel av spår en cylindrisk struktur. Det här är anledningen till att man talar om att en disk har *cylindrar*. Termen ”cylinder” betyder alltså flera våningar av spår ovanpå varandra. Och faktiskt är termen så allmän att man även referer till enskilda spår som cylindrar även om hårddisken bara skulle bestå av en skiva, alltså bara en våning.

1.3. Formatering

Den råa ytan på en disk kan dock inte användas direkt. Beroende på olika tillämpningar finns olika behov att organisera datalagringen på olika sätt, därför måste en diskyta *formateras* innan den kan användas. När det gäller CD-skivor så finns inte så många varianter, det är ISO-9660 som gäller, men när det gäller hårddiskar så finns hundratals olika filsystem. I grund och botten handlar det om saker som att bestämma klusterstorlekar och hur klustrena ska organiseras i större enheter som heter *block*. Och mera administrativa strukturer som FAT, superblock etc. Ett *kluster* är den minsta addresserbara enheten i ett filsystem och består då av ett antal sektorer. Olika filsystem har olika antal sektorer per kluster. Den grundläggande konflikten är:

Ju <u>större</u> block/kluster desto mindre antal filer att hålla reda på, desto snabbare blir filsystemet, men, desto <u>större förluster</u> i utrymmet eftersom en fil inte riktigt fyller upp stora kluster.

Omvändningen kan förstås också formuleras:

Desto <u>mindre</u> block/kluster, desto mer filer att hålla reda på, så filsystemet blir långsammare, men <u>utrymmesförlusterna blir mindre</u> eftersom det det är lättare att ge varje fil lagom med diskyta.

Olika behov finns i olika situationer. I tillämpningar med mycket stora filer, vid arkivering, till exempel, är det förstås bra med ett filsystem med stora kluster. I tillämpningar där vi har många små filer behöver vi förstås mindre kluster.

2. Partitioner

En *partition* är en cylindergrupp det vill säga en del av en hårddisk. En disk kan vara uppdelad i flera partitioner och detta är för att lättare sortera saker och ting.
På partitioner skapas filsystem genom formatering som gör att vi kan organisera användandet av utrymmet. När datorn/systemet startar sker en ihopmontering av vissa av partitionerna/filsystemen. I *UNIX/Linux* så skapas filhierarkin genom ihopmontering vid systemstart. Montering kan också ske då systemet kör. Detta har vi studerat vid installation av *Gentoo*. I *Windows* så radas partitionerna upp i form av bokstäver C:, D:, etc, (så ingen egentligen ihopmontering sker.)

Värt att notera är att ett operativsystem ofta uppfattar en partition som en egen enhet, att alltså två skilda partitioner som ligger på samma fysiska hårddisk behandlas i princip på samma sätt av operativsystemet som om de vore två skilda fysiska hårddiskar.

3. Disk Striping och Raid

Läs själva, inte så viktigt att känna till alla varianter, men det grundläggande principen är viktig att känna till: om man har flera hårddiskar som samarbetar kan man korta ner accesstiden och/eller öka lagringssäkerheten.

4. Filsystem

Ett filsystem skapas på en partition och erbjuder då ett högnivågränssnitt tilllagringsutrymmet på partitionen. Det finns olika typer av filsystem, ett för OS och ett för människan, det för människan innehåller saker som "filer", "kataloger" etc. medan det för OS är en snabb in och utskyfflare av stora mängder av det fysiska minnet.

4.1. Hierarkiska filsystem och länkar

Katalogindelningen ordnar det hierarkiska i ett filsystem, och med länkar kan man låta samma entitet (katalog, fil eller annan länk etc) förekomma på flera ställen. Den enklaste formen av länk är den så kallade *genvägen* som bara består av en angivelse av ett filnamn, en referens till en resurs på ett annat ställe i filhierarkin. *Windows* har dessa och kallar dem just *genvägar*, ”*Shortcut*” och *UNIX*-systemen har dem också, men i *UNIX*-systemen kallas de *symboliska länkar*. Skälet till detta är att det finns något som heter *länkar* i *UNIX* och för att förstå vad det är ska vi först skilja på ett *filnamn* och ett *filinnehåll*. Det är givetvis två olika saker, *filnamnet* är hur man hittar innehållet i själva filen och *innehållet* är något annat. Både filnamn och filinnehåll lagras förstås i filsystemet men det *är* principiellt två olika saker. En *genväg* är då en fil vars enda innehåll är ett annat filnamn, genvägen blir då en pekare till en annan fil som den indikerar. En genväg är då en egen självständig fysisk fil och är inte samma fil som den fil den pekar på.

I *UNIX* finns då faktiskt möjligheten att introducera *fler* filnamn till ett och *samma* filinnehåll. Vart och ett av dessa filnamn har samma dignitet som filnamn, alla är filnamn men kallas för *länkar* till filinnehållet. Vi har alltså då situationen flera namn (länkar), ett innehåll på ett fysiskt utrymme. I fallet med genvägen har vi två olika filer, genvägen själv, som är en självständig fil, och den fil som genvägen pekar på. Vidare, om man tar bort den fil som genvägen pekar på så går genvägen förstås sönder, den pekar på en fil som inte längre finns. Så här är det inte med riktiga länkar, varje länk är ett filnamn till samma lagringsutrymme, om man tar bort en länk, alltså ett filnamn, så finns de

andra länkarna, filnamnen, kvar och kan peka ut lagringsutrymmet. Det är när man tar bort det sista filnamnet, länken, som innehållet också tas bort.

En länk skapas i *UNIX* med kommandot `ln` och med kommandot `ln -s` skapas en symbolisk länk, alltså en genväg. Man brukar även kalla länkar i *UNIX* för *hårda länkar* och genvägar (alltså symboliska länkar) brukar då kallas *mjuka länkar*. Här följer en gammal tentauppgift som kan tjäna som ett exempel:

Antag att vi gör följande kommandon vid en kommandoprompt:

```
echo goddag>dagbok
cp dagbok dagbok_cp
ln dagbok dagbok_ln
echo gonatt>>dagbok_ln
ln -s dagbok dagbok_lns
rm dagbok

cat dagbok_cp
cat dagbok_ln
cat dagbok_lns
```

Vad blir resultatet?

OBS: Hårda länkar kan inte spänna över två skilda partitioner, då måste man ha symboliska länkar. Varför det? Fundera på vad som händer om man tar bort en hård länk...

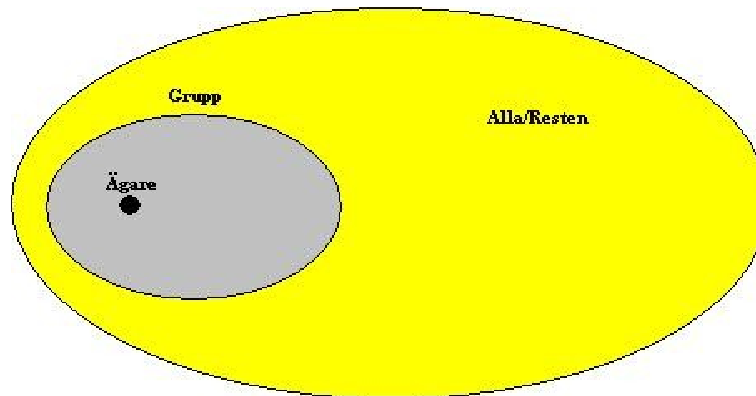
4.2. Filtyper

Ett filsystem kan innehålla många funktioner som skräddarsyr systemet mot dess användning, swap-systemen är till exempel väldigt bra på att snabbt få in och ut data, men det finns inga kataloger eller länkar. Det finns olika filsystem med rättigheter baserade på varje fil eller baserade på i vilken katalog en fil ligger. Kataloger själv har olika rättigheter och man kan koppla kontroller till om en fil kan köras eller ej baserat på rättigheter som dokumenteras av filsystemet detta kallas *Access Control Lists* och används bl.a. av *NTFS* (*Windows* filsystem). I *Windows* slutar en körbar fil på de tre bokstäverna `.exe`, som är en förkortning för ”*executable*”. I *UNIX*, däremot, som vi sett avgörs körbarheten på rättighetsbitar. Vi sammanfattar detta nedan.

4.3. Tillstånd och tillgång / Rättigheter

Som sagt, körbarheten i de vanligaste *UNIX*-filsystemen avgörs av speciella skyddbitar, det här med `rwx`, ni vet. Men det finns andra filsystem som *Andrew File System* där rättigheterna bestäms av vilken katalog man befinner sig i. Katalogbaserade rättigheter uttrycker sig alltså om hela katalogen, en rättighet kan vara att en användare får lägga till filer, läsa filer, modifiera filer etc. Läs gärna om *Access Control Lists* och *Andrew File System* på *Wikipedia*.

Vi tar en liten genomgång av det vanligaste sättet att hantera filrättigheter under UNIX.



En fil har nio så kallade "protection bits" som avgör hur ägaren, gruppen som ägaren tillhör och resten av användarna får använda en fil.

r = tillstånd att läsa
w = tillstånd att skriva
x = tillstånd att köra (exekvera)

De första tre bitarna beskriver ägarens rättigheter, de efterföljande tre beskriver gruppens och de sista tre beskriver alla andras rättigheter.

Ex:
`rwx rwx rwx`
Här har alla alla rättigheter

`rwx --- ---`
Här har ägaren alla rättigheter, men inga andra får göra någonting.

`rwx rw- ---`
Här har ägaren alla rättigeheter och gruppen får läsa och skriva i filen, men inte köra. De andra får inte göra någonting.

4.4. Filsystemsprotokoll

För att läsa och/eller skriva till en fil måste den först formellt öppnas och sedan kan läsning eller skrivning ske, därefter måste den stängas. Detta är av flera anledningar:

1. Rättighetskontroll sker vid öppning
2. Koordination vid skrivning. Öppning för skrivning medför att ett skrivlås läggs på filen.
3. processen som öppnar får fysisk information om var filen finns. (Pekare initieras.)
4. Cachning kan ske.

Olika approacher finns om en process vill skriva till en fil som en annan läser. Läs själva om dessa.

5. Implementation och lagring

Dataöverföringar blir mer effektiva om innehållet på en disk delas in i block. Överföringar sker då i hela block (kallas också kluster). Detta blir effektivare på grund av den fysiska beskaffenheten hos

en disk. Om varje enskild byte på en disk skulle adresseras och en fil skulle lagras i mycket små bitar skulle fysiken sätta ned effektiviteten. Priset vi betalar är att en fils minsta faktiska fysiska storlek blir ett block. Om ett block är tex 8KB kan det resultera i slöseri med fysiskt utrymme. Därför finns även *fragment*. (Flera fragment kan då dela på ett block.) Man kan läsa om detta i detalj i Mark Burgess kompendium, *A Short Introduction to Operating Systems*. (Finns på KTH-Social.)

Block som bygger upp en fil kan hanteras på olika sätt:

1. Länkad lista: Varje block har en pekare i sig till nästa block. Det innebär förslust av tid om filen är spridd i olika block över stora avstånd på diskytan. Man måste ju hela tiden läsa sekvensiellt.

2. Filallokeringstabell: Ett speciellt ställe av disken dedikeras till en tabell som innehåller en länkad lista av pekare ut till blcoken på diskytan. Alla blockpekare samlas på ett ställe. Denna tabell kan också cachas i *RAM*, det gör man i *Windows* (*DOS*).

3. Indexering. Detta är det vanliga sättet i *UNIX/Linux* att bygga upp filer. En fil lagras i noder som hålls ihop av en trädliknande struktur av index, de så kallade i-noderna innehåller länkar till block eller andra i-noder. Mark Burgess beskriver detta ganska bra. Läs och begrunda, ni behöver inte läsa det så jättenoga, men det kan vara bra att känna till det.

6. Översikt av olika filsystem

Vi kan bara titta på `http://sv.wikipedia.org/wiki/Kategori:Filsystem` så har vi en ganska bra översikt över en massa filsystem som det kan vara värt att känna till. Också artikeln `http://en.wikipedia.org/wiki/File_system` är att betrakta som nästan lika viktig som kurslitteraturen. Läs den, men inte för mycket.

7. Montering av partitioner, varför det?

Jag installerade *VirtualBox 4.0* innan förrförra årets kurs började. Jag använde inte *Gentoo*s `emerge` för att installera VB, jag laddade ned en binär installationsfil direkt från *virtualbox.org* och körde den, då installerades VB i katalogen `/opt`. Det är standardkatalog dit man installerar det man inte behöver för att köra systemet, *opt* är en förkortning för *optional*, det vill säga valfri.

Så långt var allt gott och väl, men jag kom senare på att eftersom jag inte valt att lägga `/opt` på en egen partition så installerades alltså en applikation som inte behövs för systemets skull i samma filsystem som rot-katalogen, alltså ”`/`”! Applikationen *VirtualBox* var ju bara för min (och er) skull och det är inte bra att blanda samman det som är högst nödvändigt för systemet (alltså det som finns i roten `/`) med det som inte alls är nödvändigt för systemet (VB). Vid en systemkrasch monteras endast rot-katalogen (`/`) och det är bra om den kan vara så skyddad och oförändrad som möjligt, då är det bra om den kan få vara ensam på sin partition och inte störas av annat. Vid drift av systemen väljer man till och med ibland att montera rotkatalogen i *read-only* för att skydda dess integritet. Det kan man göra i antingen `/etc/fstab` eller ge det som parameter till systemet vid start. Rotpartitionen, `/`, (och förstås `/home`) är systemets mest värdefulla platser och måste skyddas.

Inte bra! Hur skulle jag lösa det här? Om jag hade arbetat i ett *Windows*system och installerat en programvara på en partition som jag verkligen vill skydda så hade jag haft två val:

1. Flytta installationen manuellt genom att flytta alla installationens filer och sedan ändra om alla sökvägar som berodde på den. Ohyggligt jobbigt, dessutom är det mycket svårt att flytta om i gjorda installationer i *Windows* eftersom så mycket går in i någonting som kallas *the registry* och för att flytta om saker måste man ha specialiserad programvara.

2. Avinstallera och ominstallera.

Inga av dessa saker är så lockande. I ett *UNIX*-system är sökvägarna i filnamnen *inte* kopplade till partitionerna eftersom partitionerna monteras särskilt på kataloger och monteringen kan ändras, i *Windows* går inte det, `C:` är `C:` och `D:` är `D:` och dessa partitionsnamn är delar av alla filnamn. Eftersom filnamnen i *Gentoo* (ett *UNIX*-liknande system) inte är kopplade (eller är löst kopplade) till vilken partition de ligger på så kunde jag enkelt genom att bara flytta alla filer lägga installationen på en annan partition och göra en symbolisk länk från `/opt` till det nya stället. Det nya stället var `/usr/local` som också är ett ställe där man installerar programvara som inte är obligatorisk för systemet. Man brukar ibland välja att även ha `/usr/local` på en egen partition för att skydda `/usr`. (Det hade inte jag dock.) Allt jag behövde göra i det här läget var alltså:

1. **mkdir /usr/local/VirtualBox**
2. **cp -R /opt/VirtualBox/* /usr/local/VirtualBox/**
3. **rm -rf /opt/VirtualBox**
4. **rmdir /opt**
5. **ln -s /usr/local /opt**

Jag hade bara *VirtualBox* i `/opt` så efter jag tagit bort VB från `/opt` så blev den tom så då kunde jag bara ta bort `/opt` och lägga dit en genväg till `/usr/local` istället. Nu kör VB från `/usr/local` men alla paket och underhållsfunktioner från *Oracle* (som förutsätter att VB ligger under `/opt`) fungerar bra tack vare den symboliska länken som pekar ut var katalogen som innehåller VB egentligen ligger. Här är ett utdrag från `ls -l /`:

```
...
drwxr-xr-x       2 root root 4096 Jan  4 23:44 media
drwxr-xr-x       7 root root 4096 Jan  2 07:47 mnt
lrwxrwxrwx       1 root root   10 Jan  2 14:25 opt -> /usr/local
dr-xr-xr-x   127 root root    0 Jan  5  2011 proc
drwx------       6 root root 4096 Jan  4 23:43 root
...
```

Det här är så som ett operativsystem ska fungera om det ska fungera på riktigt! *Windows* är bra, men i grunden har *Windows* flera svagheter som gör att ett riktigt välskött *UNIX*-system helt enkelt är bättre. (Det är därför ganska gåtfullt att *Windows* är så mera utbrett. Men det finns flera saker i världen som är gåtfulla.) Låt oss beskriva det här noggrannare i *Linux* även om detta gäller för flera *UNIX*-system.

Partitionerna är enheter och representeras av enhetsnoder i katalogen `/dev`. På min dator, som jag skriver dessa anteckningar på, har finns följande exempel på partitioner och deras monteringspunkter:

> /dev/sda1 (monteringspunkt /boot), /dev/sda5 (monteringspunkt /),
> /dev/sda6 (swap), /dev/sda7 (/tmp), /dev/sda8 (/var),
> /dev/sda9 (/usr), /dev/sda10 (/home).

Det finns flera partitioner, men jag nämner inte dem här. Monteringen sker normalt vid systemstart och det defineras av filen /etc/fstab, och här är ett mer fullständigt utdrag ur den:

```
/dev/sda1           /boot         ext2          noauto,noatime  1 2
/dev/sda5           /             ext4          noatime         0 1
/dev/sda6           none          swap          sw              0 0
/dev/sda7           /tmp          reiserfs      noatime         0 2
/dev/sda8           /var          reiserfs      noatime         0 2
/dev/sda9           /usr          ext4          noatime         0 2
/dev/sda10          /home         ext4          noatime         0 2

/dev/sda11          /vir          xfs           noatime         0 2
/dev/sda12          /inst         xfs           noatime         0 2
/dev/sda4           /mnt/windows  ntfs-3g       noatime         0 0
/dev/cdrom          /mnt/cdrom    auto          noauto,ro       0 0
```

Det viktiga att se här är partionerna till vänster och monteringspunkterna i nästa kolumn. Filen /etc/fstab blir då ett sätt att koppla isär begreppet partition från begreppet filnamn, i och med att monteringspunkterna anges så kan hela systemet, när montering har skett, lugnt endast arbeta med filnamnen och behöver inte fundera över vilken partition en viss fil ligger på. Det man kan göra i en *UNIX*-miljö är ”*skifta en disk*” om det behövs, det är så här enkelt: installera den nya fysiska hårddisken, systemet kommer att tilldela den en enhetsnod, här kanske den får namnet /dev/sdb. Sedan införs ett nytt filsystem på den, kanske ext4, den får då enhetsbeteckningen /dev/sdb1. Om vi nu till exempel vill flytta vår /home till en separat hårddisk så kopierar vi först innehållet i /home in på den nya hårddisken, vi får då göra något i stil med

```
mkdir /mnt/newhome               (Skapa en tillfällig monteringspunkt.)
mount /dev/sdb1 /mnt/newhome     (Montera den nya enheten på punkten ovan.)
cp -R /home/* /mnt/newhome       (Kopiera alla filer till den nya hårddisken.)
rm -rf /home                     (Ta bort alla filer från den gamla katalogen.)
```

och sedan behöver man bara ändra i /etc/fstab till

```
/dev/sdb1           /home         ext4          noatime         0 2
```

sedan är det bara att starta om datorn, nej, man behöver inte ens göra det, man kan göra umount /home följt av mount -a som monterar allt enligt /etc/fstab. Givetvis är operationen rm -rf /home en av de mest riskfyllda. Varför det? Hur skyddar vi oss?

Eftersom *Windows* smälter ihop filnamn och partitionsnamn så är det som sagt inte lika enkelt här. I *Windows* heter alltså en till exempel C:\Documents\minfil.doc eller liknande. Flyttar man filen till en annan partition, till exempel D: så heter den något med D:\... etc etc.
Filen /etc/fstab är en av de viktigaste filerna i ett *UNIX*-system och det är alltså den som är kopplingspunkten mellan partioner och filnamnen. Varför ska vi ha en kopplingspunkt, varför inte

göra som i *Windows* och kalla filer för `/dev/sda10/johnny` istället för `/home/johnny` etc? Det är just för att vi vill ha möjligheten att separera de två aktiviteterna

1. Arbeta med filnamn
2. Arbeta med partitioner

Anledning till att vi vill separera dessa är att den första uppgiften hör till *daglig drift* av systemet, systemet kör och arbetar med filer. Den andra uppgiften hör till *systemadministration* och förenklas (som vi sett exempel på ovan) om vi kan koppla loss den från komplexiteten med filnamn. Vi ska illustrera denna viktiga princip med ett flertal liknande exempel som ni förhoppningsvis känner igen:

* I en mikrodatorarkitektur kommunicerar mikroprocessorn med minnescellerna genom ett bussystem, detta system av bussar blir då en koppling mellan processorn och minnescellerna. Vi kan lätt byta ut minnescellerna utan av behöva bryta upp processorn, och vice versa, vi kan byta ut processorn utan att greja med minnescellerna.

* I ett program, skrivet till exempel i *C*, inför vi ibland funktioner som är små underprogram som löser underuppgifter, det har ni förstås gjort, men har ni då tänkt på parameterlistan. Den bör ju vara så kort som möjligt, ju färre parametrar en funktion har, desto enklare är den att anropa och desto bättre grepp har man om den i sitt sinne och desto lösare koppling har den till det omgivande programmet i stort. Vi kan jämföra hur man skapar en process i *Windows* systemprogrammering, det gör man med anropet `CreateProcess()`, det här är dess funktionsprototyp:

```
BOOL WINAPI CreateProcess(
  __in_opt     LPCTSTR lpApplicationName,
  __inout_opt  LPTSTR lpCommandLine,
  __in_opt     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  __in_opt     LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in         BOOL bInheritHandles,
  __in         DWORD dwCreationFlags,
  __in_opt     LPVOID lpEnvironment,
  __in_opt     LPCTSTR lpCurrentDirectory,
  __in         LPSTARTUPINFO lpStartupInfo,
  __out        LPPROCESS_INFORMATION lpProcessInformation
);
```

I *UNIX*-system anropar vi funtktionen `fork()` som inte har några argument alls och följer upp anropet till `fork()` med kompletterande anrop, kanske till `exec()` etc. Men *UNIX* variant är mer modulär, alltså isärtagbar och därmed mer greppbar. Och ni kanske själva har upplevt att det är bättre med färre argument till de funktioner ni skriver.

* Vi har modularitet/löstagbarhet i anrop till funktioner, men vi skapar också modularitet/löstagbarhet i hela mjukvaruarkitekturella sammanhang genom att införa *programbibliotek*. Alla bibliotek som ni programmerat med, `stdio`, `stdlib`, `string`, etc är löstagbara och sammansättningsbara komponenter som används för att bygga programvara. Och programmen blir då lättare att underhålla. Möjligheten finns då att nämligen gå in och förbättra biblioteken (`stdio`, `stdlib` eller `string` etc.), kompilera om programmen med det nya biblioteket utan att ändra i programmen och få ett bättre program! Det här var källkodssidan, men samma sak gäller på körbara sidan, som ni numera vet finns dett något som heter *shared libraries* som ligger till grund för det som i dagligt tal brukar kallas *plugins* vilket innebär att man kan ta isär

och sätta ihop körande program under tiden som de kör. Detta möjliggörs av virtuell minneshantering och i *Windows* kallas *shared libraries* för DLL:er, *Dynamically Linked Libraries*. Ett exempel på detta är att ladda in en ny Flashspelare till en webbläsare eller på andra sätt uppdatera redan installerade program.

* Sidhantering, *paging*, är ytterligare en teknik där vi separerar logisk och fysiskt minne, en användarprocess har ingen aning om hur dess maskin är placerad i det fysiska minnet, konverteringen från logiska till fysiska adresser sköts av MMU och sidhantering blir då en koppling mellan två begrepp som vi gärna vill koppla isär: Hur en processbild härbärgeras i minnet (de fysiska adresserna) och hur den processen själv kör internt (de logiska adresserna).

Vi kan fortsätta och räkna upp massor av exempel där det alltså är mycket önskvärt med löstagbarhet, eller isärtagbarhet... det finns faktiskt ett formellt datortekniskt begrepp som definierar just detta och det är *coupling*, alltså *koppling*, inom datorteknik vill vi ha låg *coupling* alltså låg koppling mellan komponenter inom datortekniken.

En utmärkt formulering (och också bra definitioner av *coupling* och *cohesion*) av detta finns på ”http://www.hokstad.com/why-coupling-is-always-bad-cohesion-vs-coupling.html”. Ett utdrag därifrån är ”**Coupling is always bad because it prevents the replacement or changes of components independently of the whole.**”

Ett annat exempel på detta är debatten om mikrokärnor och monolitkärnor i operativsystemtekniken. Ytterligare ett litet exempel på någonting med låg koppling kan vara en symbolisk länk och det den pekar på. (Hela *Client-Server*-begreppet kan också ses som en strävan efter låg *coupling*.) I fortsättningkurser i objektorientering kommer ni också att stöta på detta begrepp, vi kommer också att se det i senare delar av denna kurs och rent allmänt kan man säga att ni kommer att se det i hela ert arbetsliv som datoringenjörer.

8. *UNIX* mer utvidgade syn på filer

I ett *UNIX*-system kan man komma åt väldigt mycket via filsystemet. Hierarkin under rotkatalogen (`/`) innehåller förstås filer som har ett underliggande lagringsmedium som hårddiskar, CD-skivor, USB-minnen mm, men det finns också filer i hierarkin som inte har något underliggande externs lagringsmedium. Vi ska se på två stycken, `/proc` och `/dev`. Kommandot `man fs` ger oss en bra översikt och referenser till andra intressanta manualsidor.

8.1 Filsystemet procfs, /proc

Kommandot `man proc` ger oss följande manualsida (bara början är angiven):

```
NAME
       proc - process information pseudo-file system

DESCRIPTION
       The  proc  file system is a pseudo-file system which is used as an inter-
       face to kernel data structures.  It is commonly mounted at  /proc.   Most
       of it is read-only, but some files allow kernel variables to be changed.

       The following outline gives a quick tour through the /proc hierarchy.

       /proc/[pid]
              There  is  a  numerical subdirectory for each running process; the
```

```
subdirectory is named by the process ID.  Each  such  subdirectory
contains the following pseudo-files and directories.
```

Filsystemet `procfs` är alltså ett så kallat *pseudofilsystem*, det har inget underliggande lagringsmedium utan är en inblick i kärnan på ett körande system. Det finns kataloger och filer under `/proc`, men dessa är snarare kontaktpunkter med de körande processerna. Varje katalog representerar en körande process som vi ser ovan. Kommandot `ps` läser ur `/proc` och baserar sitt resultat på innehållet i `/proc`.

8.2 Filsystemet under `/sys`, (`sysfs`)

Under senare år, i alla fall i *GNU/Linux*-världen, har systemet under `/proc` blivit ganska belastat av saker som inte riktigt hör dit, bland annat saker som hört mer till de inkopplade kringenheterna. I och med 2.6-serien av linuxkärnan har man därför börjat lägga in mer saker i ett annat system som heter `/sys`. Vi kommer troligen inte att studera så mycket vad detta konkret innebär, men vi kan faktiskt här också observera begreppet coupling igen. Man lade in saker i `/proc` som inte riktigt hörde hemma där, det skapade hög koppling mellan saker som opererar på periferienheter och systemet `/proc` samtidigt som `/proc` skulle sköta andra saker. Systemet `/proc` fick då två roller, dels att presentera körande processer i filform och dels att presentera information om det körande systemets enheter i filform. Åtgärden som minskade denna koppling var att införa `/sys` som alltså sorterar saker på ett bättre sätt och medför lägre koppling till systemet `/proc`. För mer information om det här, läs gärna http://en.wikipedia.org/wiki/Sysfs.

8.3 Filsystemet under `/dev`

Filerna under katalogen `/dev` är vanligtvis inte heller filer med underliggande lagringsmedium, de är oftast kontaktpunkter till olika enheter som är anslutna till datorn. Vi har förstås stött på partitionerna som i *SCSI*-utförande har utseendet `/dev/sda1`, `/dev/sda2`, `/dev/sda3` osv under *Linux* och under *BSD* har de utseendet `/dev/ad0s1a`, `/dev/ad0s1b` osv. Device betyder ju enhet på svenska och filerna under `/dev` är som sagt kontaktpunkter till olika enheter, men inte bara periferienheter som diskar, pekdon etc, det finns också virtuella terminaler, `ttyv0`, `ttyv1` osv som representerar in- och utmatningsterminaler som används när man loggar in i textläge. Vi har också så kallade *pseudoenheter* (*pseudo devices*) som

`/dev/null`	–	en inenhet som bara tar bort det man skriver till det.
`/dev/zero`	–	en utenhet som bara matar ut nollor.
`/dev/random`	–	en utenhet som matar ut slumptal.

Under `/dev` finns också de viktiga kategorierna *block devices* och *character devices*. Det får ni läsa om vad det är själva, vi kan nämna att vi stött på block devices, partitionerna på en hårddisk är nämligen sådana.

8.4 Andra typer av filer

Vi har nu sett en massa sorters filer, *reguljära filer*, de med ”–” först i sin `ls -l`-listning, kataloger, de med ”`d`”, symboliska länkar, de med ”`l`”. Character devices har ”`c`” och block devices har ”`b`”. Men det finns flera! Man kan skapa något som heter en *named pipe* eller *fifo* vilket är en

pipe fast den existerar i filsystemet. Man skapar den med `mkfifo` och kan sedan läsa och skriva till den som en vanlig pipe, den har ett ”p” som första bokstav i sin `ls -l`-listning. Sockets kan också läggas in i filfilhierarkin, de får då ett ”s”. Detta är då UNIX-domain sockets. Om ni gör `ls -l` i katalogen `/dev` så hittar ni nog exempel på alla dessa. Experimentera också gärna med `mkfifo` enligt följande:

`mkfifo minpipe` (gör en pipe)
`cat < minpipe &` (starta en `cat`-process som läser från `minpipe` och skriver ut på skärmen)
`echo hej hej>minpipe` (skriv något till pipen och se vad som händer)

För tydlighetens skulle kan det vara bra att göra skrivningen (`echo …`) i ett terminalfönster och läsningen (`cat < minpipe`) i ett annat.

8.5 *UNIX* / *POSIX* kombinationsmöjligheter

Vi har ovan studerat begreppet *coupling*, alltså koppling, och funnit att det inte är bra. Någonting som vi även vill ha förutom låg koppling i ett datorsystem är kombinationsmöjligheter, vi vill kunna kombinera olika funktioner på alla tänkbara sätt. Detta kallas med ett fint ord ”ortogonalitet”. *UNIX* understödjer detta bland annat genom att se allting i en stor filhierarki, alla filer där kan normalt kombineras med varandra och verktygen i systemet på alla tänkbara sätt. Det är alltså helt OK att tänka sig att läsa från en fil, skicka resultatet via en pipe, eller en socket, mixa in lite sortering och därefter skicka på en annan pipe eller socket till det ena eller det andra. Den tredje laborationen uttrycker detta lite grann, där vi studerar hur flera barnprocesser kommunicerar via två pipes och där vi byter ut en av piparna mot en socket. Fildeskriptorer i *UNIX* kan, via omdirigering peka på vilka filer som helst. Det är bland annat det här som *UNIX* har som designfilosofi: *tools not policy*.

Systemadministration samt något om kompilering och interpretering

1. Ett ödmjukt försök till ödmjukhet...

Det här dokumentet (alltså dessa anteckningar) har kommit till på ett speciellt sätt, jag hade två installationer på min dator, en Windows XP (jag har slutat använda Windows 7), och Gentoo Linux som hade tre virtuella installationer av Winodows XP, FreeBSD och Windows XP. Jag försökte flytta Windowsinstallationen till en annan partition än den den låg på och då kraschade allting. Det illustrerar en viktig poäng: man måste göra säkerhetskopior innan man börjar arbeta med partitionerna på en dator. Jag hade säkerhetskopierat det viktigaste och jag gjorde sedan en ominstallation av Gentoo och Windows XP. Sedan gjorde jag även om de virtuella installationerna av Gentoo, FreeBSD och Windows XP. Det illustrerar också en annan sak: var försiktig med att flytta Windows!

Så, frågan är ju om jag är rätt person att undervisa om systemadministration om jag klantar mig på det här sättet? Jag tror att vi måste anta en tolerant och ödmjuk attityd mot oss själva när det gäller datalogi och datorteknik, ingen enda människa kan ha djupa kunskaper om allt som pågår inom databranschen. Så säkerligen finns det andra som är bättre än jag, säkert flera här inne och jag bjuder också in de som har detaljkunskaper att gärna komplettera då de ser att någon kunskap saknas i min framställning av operativsystemteori och teknik. Det är bara varmt uppskattat.

2. Systemadministration i allmänhet

Nog med ursäkter, nu ska vi beskriva systemadministration så gott det går. En av de viktigaste administrationsuppgifterna är förstås att skapa en grundläggande operativsysteminstallation och det har ni redan gjort, ni har installerat Gentoo (inte det enklaste!) och organiserat om en installation av Debian.

De flesta OS-installationer följer just den gång som ni tränar på i och med att ni arbetar med OS-installation så har ni fått en uppfattning om hur ett operativsystem är uppbyggt i relation till sin omgivning, alltså hårdiskarna, nätverkskorten, skärmar, tangentbord etc. Systemadministration går förstås ut på att ett operativsystem ska upprätthålla och utveckla goda relationer med sin omgivning och förstås användarna av systemet. Följande uppgifter finns då:

0. Den första uppgiften, att installera och konfigurera själva systemet. Det har ni som sagt provat på redan.

1. Backup, att ta säkerhetskopior av viktiga data. Det är en uppgift som man kan schemalägga att köra över natten när ingen använder systemen.

2. Installation av ny programvara och uppdateringar till gamla programvaror.

3. Reparera saker som går sönder. Det kan innebära avinstallation följt av ominstallation. (Det som jag fick göra.)

4. Hantera systemets användare, det vill säga lägga till och ta bort användare vartefter behoven uppstår. Här ingår också att utöka eller inskränka olika användares rättigheter varefter de behoven varierar.

5. Skydda systemet (eller systemen) mot obehöriga intrång. Det här är förstås en hel kurs (eller utbildning!) i sig, hur hanterar man till exemepel säkerheten i ett nätverk med dagens alla påhittiga personer som tar sig förbi alla möjliga säkerhetsarrangemang.

Här har jag beskrivit uppgifterna hos en systemadministratör som kanske arbetar på en stor organisation med många användare, troligtvis finns då också ett team, "systemfolket" som ofta tillkallas då det är problem. Jag har en viss beundran för systemfolket, de har många krav på sig, dels ska de ha en mycket hög teknisk kompetens och sedan ska de klara av att prata med många olika sorters människor, och när en människa har problem med något (och det är då man pratar med systemadministratören) så är den människan kanske inte så lätt att prata med. Jag tror att vara systemadministratör är ett av de tuffare jobben.

Jag tycker att ett ord som sammanfattar mycket med operativsystemadministration är "integritet". Ordet "integritet" kan betyda många saker, helhet, säkerhet, pålitlighet och det är just dessa egenskaper som vi vill att ett datorsystem/operativsystem ska ha.

3. Hur genomförs systemadministration?

En sysadm måste ofta genomföra förändringar som kräver rättigheter att ändra på systemets driftmässiga inställningar, i UNIX-världen måste man då använda användarnamnet root som är den användare som har alla rättigheter. I Windows heter denna användare Administrator. I allmän operativsystemteori brukar man kalla denna användare för "superuser".

Man ska alltså använda ett administrationskonto. Hädanefter kommer jag att säga "root" om detta även om det jag säger också gäller Windows.

Eftersom root-kontot har alla rättigheter finns det stora risker för skador om sysadmin gör fel. Det betyder att sysadmin självfallet måste vara väldigt kompetent men också att han/hon måste få tillräckliga resurser, annars kan han/hon inte utföra sitt arbete tillfredställande och det blir hela organisationen lidande på. Det är väldigt viktigt med integritet och pålitlighet i alla mänskliga sammanhang, men det är särskilt viktigt för en systemadministratör. Integritet kan uttrycka sig så här, rätt sak på rätt plats: En sysadmins uppgifter får inte utföras med ett vanligt användarkonto och det enda som root-kontot ska användas till är sysadmins uppgifter. Detta kan också sägas vara ett uttryck för integritet, allt ska med, helhet, och vara på rätt plats, pålitlighet.

4. Att rädda system

En speciell färdighet som en sysadm måste ha är att kunna rädda system. Olyckor sker och data – det mest värdefulla som finns i ett operativsystem – går förlorade. Det är mycket stora olyckor då data går förlorat. Ofta kan data räddas genom att det backas upp, men ibland kraschar hela system. Jag beskrev det ovan, det jag försökte göra var att flytta en Windowsinstallation till en annan partition. Linux går utmärkt att installera på en partition och sedan bara flytta till en annan partition, man kan meka med filen /etc/fstab (som bestämmer hur partitionerna ska monteras vid systemstart) och ominstallera bootladdaren, det fungerar. Men det fungerar inte lika bra med ett Windowssystem. Det vill inte starta. Det jag gjorde var att flytta Windowsinstallationen från /dev/sda1 (första primära partitionen) till /dev/sda4 (sista primära partitionen. Jag genomförde flytten med ett verktyg som heter PartImage, som finns på SystemRescueCD, vårt fina Linux-verktyg, och försökte ställa om i bootladdaren så att Windows skulle starta på sitt nya ställe, men det gick inte. Jag gjorde då misstaget att aktivera en funktion i GRUB (GRand Unified Boot-loader) och valde att gömma de tre första partitionerna så att Windows skulle få för sig att det var ensamt på datorn. Det fungerade inte heller och när jag tog fram partitionerna igen var den utökade partitionen förstörd. I den utökade

partitionen hade jag /, /var, /usr, /home (och /tmp och swap). (/boot låg som primär partition utanför. Samtliga partitioner i den utökade partitionen var sönder.

Jag försökte då göra som man ska göra när ett filsystem krånglar, man använder fsck på de enheter som krånglar. Jag startade alltså SystemRescueCD och körde

```
fsck.ext4 /dev/sda5 (det var /)
fsck.ext4 /dev/sda8 (det var /var)
fsck.ext4 /dev/sda9 (det var /usr)
fsck.ext4 /dev/sda10 (det var /home)
```

men det gick inte. Skadan var svår att reparera. Det hade kanske kunnat gå om jag varit mer envis, det normala jag skulle gjort här var att helt enkelt googla på några av de felmeddelanden som jag fick, men jag satt i en stuga i skogen när jag skrev detta och jag hade ju säkerhetskopierat det viktigaste så jag kunde faktiskt bara kasta operativet åt sidan och göra en ominstallation. Vid det här tillfället var det också lämpligt att skriva en föreläsning om systemadministration, jag vill berätta om mina erfarenheter så att ni kan få nytta av dem.

5. Design för underhåll

Jag kunde kasta operativsystemet skriver jag ovan, det möjliggjordes av att jag som sagt hade säkerhetskopierat det viktigaste, mina filer, mina dokument (nämligen den här kursens föreläsningsanteckningar.)

Här blir det då lätt att se motiveringen till varför man har olika partitioner, att ha olika partitioner är ett led i att man bygger upp sitt system _så att det är lätt att underhålla_. Med en separat partition för /home är det lätt att med bara några handgrepp (och verktyget PartImage) dra en säkerhetskopia av samtliga dokument för samtliga användare i hela systemet.

På samma sätt kan man dra säkerhetskopior av delar av filhierarkin, /boot, /usr etc. och reparera bara det som gått sönder. Jag har tidigare använt PartImage för att dra säkerhetskopior av hela systeminstallationer, då har jag kunnat ta en hel säkerhetskopia av ett helt system, tömma datorn helt och sedan bara kopiera tillbaka systemet via funktionen restore. Jag var dock tvungen att ominstallera boot-laddaren, Grub, men annars var det inga bekymmer, Läs gärna manualsidan om PartImage genom att skriva partimage --help vid en prompt på SystemRescueCD.

När man bestämmer vilka filsystem man ska installera på olika partitioner tar man hänsyn till sådana här saker: Innehållet i /var är ett som varierar mycket, det innehåller många små filer, filsystemet reiserfs anses vara ett bra val för det. Då kan man formatera just den partition som ska innehålla /var med reiserfs. XFS är ett annat filsystem som anses vara bra för stora filer, på en partition formaterad med XFS är det alltså lämpligt att lägga saker som virtuella hårddiskar och, förstås, säkerhetskopior av andra partitioner.

Med "design" menas hur man tänker sig en uppbyggnad av något, och när vi ska skapa en operativsysteminstallation behöver vår design av installationen vara anpassad till vad systemet ska användas till. Jag har genom åren av den här anledningen kommit att fastna för just den här konstellationen (det är den vi använder i installationen av Gentoo/Arch, förutom att vi inte har sda3 respektive sda4):

Första primära partitionen, /dev/sda1, monteras på /boot. (ext2)

Andra primära partitionen, /dev/sda2, är en utökad partition som innehåller:

Första logiska partitionen, /dev/sda5, monteras på /. (ext4)
Andra logiska partitionen, /dev/sda6, är swapspace. (swap)
Tredje logiska partitionen, /dev/sda7, monteras på /tmp. (ext2)
Fjärde logiska partitionen, /dev/sda8, monteras på /var. (ext4)
Femte logiska partitionen, /dev/sda9, monteras på /usr. (ext4)
Sjätte logiska partitionen, /dev/sda10, monteras på /home. (ext4)

Tredje primära partitionen, /dev/sda3, kanske arkiv (XFS), kanske Windows (ntfs)

Fjärde primära partitionen, /dev/sda4, kanske arkiv (XFS), kanske Windows (ntfs)

(Jag hade inte ovanstående konstellation när jag begick mitt misstag, jag hade Windows på /dev/sda1 och ville som sagt flytta den till /dev/sda4.)

Man kanske skulle kunna tänka sig att lägga / och /boot på samma primära partition, /dev/sda1, det är så BSD brukar göra och kanske montera den som read-only. Det finns ju (tydligen) en viss risk med att ha / som logisk partition. Vidare kanske man av samma skäl vill ha /home på en primär partition.

Mitt mål var att ha de två primära partitionerna i formatet ntfs och ha Windows XP på /dev/sda3. Den blir då C: under Windows. Om allt går bra så ska då /dev/sda4 blir D: eller E: under Windows. Det är ju så som Windows hanterar parititioner, det är C: och D: och E: etc. De monteras inte ihop i någon struktur per default.

I UNIX/Linux/POSIX är montering något som gjorts enkelt på så sätt att systemet håller reda på hur applikationer ska komma åt hårddiskar, I Windows om man vill ändra hårddiskarna placering/numrering måste man antingen installera om alla programvara som beror på en viss sökväg, eller gå in och ändra sökvägen överallt där den förekommer, inga av dessa varianter fungerar i ett professionellt sammanhang. (Ja, det har vi ju talat om i tidigare föreläsningar.)

6. Användarhantering under UNIX

Vi ska särskilt nämna några ord om användarhanteringen under UNIX eftersom den har beröring med många områden i systemadministration.

6.1 Rättighetssystemet i UNIX

Som ni kanske vet baseras rättighetssystemet i UNIX på användaridentitet och grupptillhörighet. En användare kan vara medlem i flera grupper men har en huvudgrupp som han/hon är medlem i. Vad en användare sedan får göra med filerna bestäms av vem användaren är och vilka gruppern han/hon är medlem i. Vi tar ett par exempel:

Vi har följande katalogstruktur givna av kommandot ls -l:

```
total 24
-rw-r----- 1 johnny johnny 10799 Dec 30 13:29 FL-OS-Sysadm.txt
drwxr-xr-x 2 johnny johnny  4096 Dec 30 13:10 bin
drwxr-xr-x 3 johnny johnny  4096 Dec 30 13:08 kurser
drwxr-xr-x 2 johnny johnny  4096 Dec 30 13:07 vatisen
```

Det här är /home/johnny på min bärbara jobbdator. Filen FL-OS-Sysadm.txt har rättighetsprofilen -rw-r-----. Det första strecket betyder att det är en reguljär fil, de undre filerna har d som första tecken i rättighetsprofilen vilket innebär att de är kataloger (alltså filer som innehåller andra filer). De åtföljande 9 teckena kallas “protection bits” och uttrycker vad allas rättigheter i hela systemet är, gällande filen FL-OS-Sysadm.txt. Vi ser på det tre sektionerna, **rw- r-- ---**. De första tre bitarna (**rw-**) gäller användaren som alltså har läs och skrivrättigheter, men inte rättighet att köra filen. Filen är inte körbar, den innehåller texten till denna föreläsning så det är inget script eller program som det meningsfullt att köra. De nästföljande tre bitarna (**r--**) gäller alla medlemmar i gruppen “johnny”. Nu är johnny en användare, men en vanlig teknik är att också skapa johnny som en grupp. Det innebär att gruppmedlemskap i gruppen “johnny” kan delas ut till andra användare som få del av del rättighetsprofil som gäller filer som johnny äger. Det finns också en annan möjlighet och det är att låta grupprättigheter till en fil styras av en grupp som inte är kopplad till just användaren johnny, en annan variant vore att ha gruppen users rättigheter kopplade till dessa filer, då skulle listningen kanske se ut så här:

```
-rw-r----- 1 johnny users 10799 Dec 30 13:29 FL-OS-Sysadm.txt
drwxr-xr-x 3 johnny users  4096 Dec 30 13:08 kurser
drwxr-xr-x 2 johnny users  4096 Dec 30 13:07 vatisen
```

Det finns lite olika effekter av detta, vi ska inte jämföra så noggrannt just här, men ni ska veta att möjligheterna finns.

Vi gör avslutningsvis bara observationen att alla andra (others) har protection bits satta till **---**, vilket innebär att andra som inte är johnny eller medlem i gruppen johnny (eller users om man väljer den vägen) inte har några rättigheter alls på den aktuella filen.

Rättighetsbitarna kan ändras av ägaren (eller root) med kommandot **chmod**. Filtillhörighet (vem som äger vilken fil) och grupptilhörighet hos enskilda filer ändras av root med kommandot **chown** respektive **chgrp**. För att se hur dessa kommandon fungerar, se manualsidorna.

6.2 Sticky bit, SUID och GUID-bitarna

Det som inte syns så väl av listningen ls -l är att det faktiskt finns ytterligare tre bitar som vi kan visualisera vill vänster om de nio vanliga protection bits. Det är SUID, GUID och sticky bit. Vad är detta för djur då? Jo, biten SUID, som är mest signifikant fungerar så här, om den är satt så får filen köras med de rättigheter som ägaren till filen har. Det normala är att rättigheterna hos den som kör igång programmet bestämmer vad programmet får göra. Vi tar ett konkret exempel för att illustrera principen. Vi betonar att det bara är ett skolexempel, vi ska inte göra så här på ett system i drift.

En vanlig användare får inte skapa kataloger i roten utan normalt bara i sin egen hemkatalog. Om jag, inloggad som johnny, skriver

mkdir /testdir

får jag alltså ett felmeddelande:

mkdir: cannot create directory `/testdir': Permission denied

Som vanlig användare har jag alltså inte skrivrättigheter i rotkatalogen, vilket

ju är naturligt.

Om vi skriver **which mkdir** får vi upplysning om att mkdir-kommanot som används är /bin/mkdir. Om vi då skriver **ls -l /bin/mkdir** så får vi följande resultat:

-rwxr-xr-x 1 root root 47496 Dec 23 10:47 /bin/mkdir

alltså root äger kommandot och vanliga användare, som johnny, har körrättigheter på kommandot. Men när användaren johnny kör kommandot så körs det med johnnys rättigheter, när johnny alltså försökte göra **mkdir /testdir** så nekas han det eftersom han inte har skrivrättigheter i rotkatalogen. Vi ska absolut inte se det här som ett problem! Det är precis som det ska vara, användaren johnny, ska absolut inte ha skrivrättigheter i rotkatalogen! Men vi ska illustrera vad SUID-biten är för något. Då gör root följande kommando:

chmod u+s /bin/mkdir (alltså sätt SUID-biten till 1 för /bin/mkdir)

och resultatet av **ls -l /bin/mkdir** är nu

-rwsr-xr-x 1 root root 47496 Dec 23 10:47 /bin/mkdir

och vi ser att vi har ett **s** på platsen där vi tidigare hade ett **x**. SUID-biten är satt och när nu användaren johnny kör kommandot mkdir så körs det med roots rättigheter, eftersom root har skrivrättigheter i rotkatalogen så kan vem som helst som får köra kommandot mkdir skapa kataloger var som helst i systemet.

Vi ser på ett utdrag ur ett par kommandon:

```
johnny@porthos ~ $ mkdir /testdir
johnny@porthos ~ $ ls -l /
total 60
drwxr-xr-x   2 root root   4096 Dec 29 17:13 bin
drwxr-xr-x   2 root root   4096 Dec 25 02:32 boot
drwxr-xr-x  16 root root   4160 Dec 30 12:55 dev
drwxr-xr-x  62 root root   4096 Dec 30 13:55 etc
drwxr-xr-x   4 root root   4096 Dec 30 13:20 home
drwxr-xr-x   2 root root   4096 Dec 25 02:33 inst
lrwxrwxrwx   1 root root      5 Dec 29 13:47 lib -> lib64
drwxr-xr-x   3 root root   4096 Dec 29 15:43 lib32
drwxr-xr-x  10 root root   4096 Dec 29 17:13 lib64
drwxr-xr-x   2 root root   4096 Dec 29 15:46 media
drwxr-xr-x   4 root root   4096 Dec 23 05:14 mnt
drwxr-xr-x   2 root root   4096 Dec 23 02:40 opt
dr-xr-xr-x 121 root root      0 Dec 30 13:55 proc
drwx------   5 root root   4096 Dec 30 15:00 root
drwxr-xr-x   2 root root   4096 Dec 29 16:41 sbin
drwxr-xr-x  12 root root      0 Dec 30 13:55 sys
drwxr-xr-x   2 root johnny 4096 Dec 30 15:00 testdir
drwxrwxrwt  10 root root    368 Dec 30 13:35 tmp
drwxr-xr-x  15 root root   4096 Dec 29 16:28 usr
drwxr-xr-x  15 root root    368 Dec 29 16:41 var
drwxr-xr-x   2 root root   4096 Dec 25 02:33 vir
```

Observera att det här systemet nu **inte** är i ett bättre tillstånd! Det är inte bra att vem som helst har “friheten” att skapa katalogen var som helst! Systemet är sönder och måste reperas genom att SUID-biten nollställs och skräpkatalogen tas bort. Det åstadkoms genom att root gör kommandona **rmdir /testdir** för att ta bort katalogen som skapades och **chmod u-s /bin/mkdir** för att återställa

rättigheterna på kommandot /bin/mkdir. (Observera ovan att ägaren till katalogen /testdir/ blen root men gruppen blev johnny. Vidare skulle inte johnny kunna ta bort katalogen /testdir eftersom rmdir inte har en satt SUID-bit.)
Med SUID kan man alltså dela ut rättighet att köra enskilda kommandon till alla användare. Observera att detta inte fungerar med shellscript! Varför inte det?

Vi kan alltså inte förvänta oss att scriptet

```
#!/bin/bash
mkdir $1
```

fungerar som ovan om vi sätter SUID-biten på det. Varför...

Eftersom det inte fungerar med shellscript så ser jag en annan möjlighet och det är att bädda in det kommando man vill utföra i ett C-program som utför kommandot som ska utföras med systemanrop i C och sedan sätta SUID-biten på det programmet. Det fungerar och vi ser ett exempel på detta på nästa sida.

Ni får själva läsa om vad Sticky-bit och GUID är i manualsidor och på nätet.

Läs vidare även själva om installation av programvara. Det kommer inte att examineras, men är viktigt att känna till.

Ett C-program som SUIDat och lagt i /bin alltså möjliggör för vanliga användare att köra mkdir. Det är skrivet i systemprogrammeringsstil där vi hela tiden kontrollerar de systemanrop som vi kör:

```
#include <sys/stat.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>

main(int argc, char *argv[])
{
  int result;

  if(argc!=2)
  {
    fprintf(stderr,"%s: usage: %s DIRECTORY\n", argv[0], argv[0]);
    exit(1);
  }

  result = mkdir (argv[1], 0x1ff );

  if(result!=0)
  {
    fprintf(stderr,"%s: could not create directory %s\n", argv[0], argv[1]);
    exit(2);
  }

}
```

det här programmet fungerar om man SUID:ar det, dvs låter ägaren vara root och sätter SUID-biten (utan att mkdir SUID:as), men det är inte någon bra ide förstås. (Av de skäl som beskrivits ovan.)

7. Något om kompilering och interpretering

Nu har vi ett perspektiv på hur program kör, de kan vara script eller kompilerade program. Kompilerade program består av maskinkod som kör direkt på systemets processor medan interpreterade program, så kallade script, behöver en tolk som tolkar programmen då de körs. Det går förstås långsammare med interpreterade program men det finns en del fördelar med interpretering, det är lättare att redigera ett interpreterat program eftersom man inte behöver kompilera det. Många av scriptspråken, som bash och perl och python etc. lämnar även över mycket till den tolk som de kör så det är inte så svårt att programmerera i dessa språk. Tolken innebär också ett slags skydd för systemet.

7.1 Java

Ett annat mycket stort och troligen världens viktigaste interpreterade språk är Java (och dess Microsoft-variant, C#). Språket är interpreterat men det finns ändå en kompilator som kompierar javakoden till någonting som kallas byte-kod. För att kunna köra ett javaprogram måste man sedan ha installerat en javatolk “Java Virtual Machine” på den plattform som ska köra javaprogrammet. Ni kommer att lära er mycket om det i 2:an då ni kommer att lämna C och gå helt över till Java som huvudspråk. (I 3:an kommer ni troligen att varva med en massa olika språk som Java, C#, C, SQL etc.)

Java kompileras som sagt till byte-kod och det är den som är plattformsoberoende, vi kan ta bytekod framställd under UNIX och köra den på en Windowsmaskin eller tvärtom. Vidare finns ett koncept som heter HotSpot-kompilering vilket, hävdar några, nästan tar bort nackdelen med att vara ett kompilerat språk (alltså att det går långsammare att köra.) I O'Reilly-boken “Learning Java” hävdas det till och med att Javaprogram blir **snabbare** än kompilerade program tack vare HotSpot-kompilering. Jag tror att man måste precisera sig där (och jag tror att de också gör det i den boken) och berätta vilken typ av applikationer man menar när man säger något sådant här.

Internet-domain sockets i C, grundläggande IP och Client/Server-applikationer

I föreläsningen gavs en mycket kort översikt till kommunikation via nätverk, vi ska nu se noggrannare på den tekniska bakgrunden till denna kommunikation. Vi ska först beskriva en mycket viktig övergripande teknik *Client-Server*begreppet. Vi kommer sedan att fortsätta berätta om grundläggande IP, sedan kommer vi att beskriva hur IP fungerar i *C* och vi kommer då speciellt att se på så kallade *sockets* som är *BSD*:s (och *POSIX* och därmed större delen av *UNIX*-världens) sätt att tillhandahålla nätverksprogrammering, det är denna kommunikationsarkitektur som det stora nätet bygger på. En av de viktigaste tekniska uppfinningarna genom tiderna.

1. The Client-Server model

För att tillhandahålla tjänster (*Services*) introduceras så kallade *servrar*. (sing. server, eng. servers) Det är engelska för betjänt. Den (eller det) som betjänas kallas klient. (eng. client). *Client-Server* begreppet är väl utvecklat. Här följer några välbekanta exempel:

Client	*Server*
Webbläsare	Webbserver
dhcpklient	dhcpserver
Fildelningsprogram	Filserver
Ordbehandlare som vill skriva ut	Skrivardemonen
Mailläsare	Mailserver
Putty (SSH-client)	SSH-server

På sätt och vis kan också de körande processerna i ett operativsystem anses vara klienter till operativsystemets kärna. (Kärnan blir så en server till processserna.) Både servern och klienten är processer och i Multitasking-, Multiuser-Operating Systems kan de finnas var som helst på ett nätverk. Det är dock vanligast att man dedikerar en eller flera datorer åt uppgiften att köra serverprocesser. Dessa datorer brukar då också kallas servrar vilket på sätt och vis är fel, en server är ju en *process*, eller i alla fall program som kan innehålla flera samverkande processer, inte en dator. En *server* är alltså, noggrannare uttryckt, en process (program) som förfogar över vissa resurser som ska fördelas till andra processer som då kallas den serverns *klienter*.

Motiveringen till client-servermodellen är att vi vill centralisera vissa specialiserade tjänster till maskiner som är skräddarsydda för att tillhandahålla dessa specialiserade tjänster, maskinerna kallas då som sagt *servrar*, de kör serverprocesser, och tjänsterna som tillhandahålls kan vara webbpublicering (webbserver), databashantering (databasserver), fillagring och tillhandahållning (filserver) etc. Fördelen är att en server då samordnar tillhandahållandet och underhållet/administrationen till en punkt, alltså en datormaskin. En filserver på skolan här tillhandahåller till exempel allas hemkataloger, H:, och dessa tillhandahålls över nätet och man får ju H: monterad var man än loggar in på de stationära datorerna. (Nu har inte ni jobbat så mycket stationärt.) (Tänk efter på begreppet *coupling*, hur reducerar client-server-tekniken koppling?)

En serverprocess är ofta flertrådad. Varför det? Kommunikation mellan klienter & servrar kräver IPC & protokoll, denna inter-process kommunikation ska gå över ett nätverk varför protokollet måste möjliggörna kommunikation över nätet, det är detta protokoll som kallas *IP*.

2. IP – Internet Protocol

Internet, och många mindre internet, bygger på IP och man hör ibland även förkortningen "TCP/IP". TCP står för *Transmission Control Protocol* och används tillsammans med IP för att möjliggöra en mer avancerad form av kommunikation mellan noder på nätet, men låt oss inte ta det nu.

Vi skriver "noder på nätet" och inte nödvändigtvis datorer, det finns nämligen andra apparater än datorer som man ansluter till nätet, främst så kallade *routrar* och *switchar*. Vi kommer inte att studera det i denna kurs, men vi kommer att använda ordet noder som är mer precist eftersom det då tillåter oss att referera till maskiner på nätet än just bara datorer.

Som omnämnt i föreläsning 1 möjliggör IP kommunikation mellan noder på nätet genom att varje nod tilldelas en IP-adress. Nu ska vi se djupare på hur det går till och vad det betyder.

2.1. Protokolltyper och datainkapsling inom nätverkskommunikation
En mycket viktig teknik inom datortekniken och speciellt inom nätverkskommunikation är datainkapsling och protokoll. Det gäller också vid kommunikation över nätverk. Varje nod kan ha viss beräkningskapacitet och kan vara en dator eller olika instrument för att förbinda datorer (switch/router etc.)

Vi har olika nivåer i datorkommunikation:

1. Fysiska nivån. När man vill förbinda två datorer med varandra tar man en sladd och stoppar in i nätverkskorten. Då har vi en fysisk förbindelse mellan två noder. Man kan också ansluta de båda datorerna till en switch (eller hub) och på så sätt förbinda fler än två datorer med varandra. Den fysiska förbindelsen kan upprättas mellan flera noder än två. Detta kallas den *fysiska nivån*. (Physical Layer.) Den fysiska förbindelsen möjliggör kommunikation på nästa nivå.

2. Lokala nätverksnivån. När nätverkskorten på noderna känner av att de är förbundna (och noderna är rätt inställda) så kan de sända data till varandra i form av *paket*. Ett paket är en följd av signaler, ettor och nollor, som har ett visst mönster. Paketet blir den grundläggande databäraren i ett nätverk. För att lyckas med detta måste noderna känna till de fysiska adresserna på nätverkskorten. En fysisk adress är någonting som är inbränt permanent på ett nätverkskort och digitaltekniken i ett nätverkskort känner av då ett paket är adresserat till det kortet och kopierar då in paketet till sin buffer. Alla noder känner till alla andra noders fysiska adresser inom samma lokala nätverk. Kommunikation på denna nivå går då till så här: Eftersom nod 1 är ansluten till nod 2 och nod 1 känner nod 2:a adress så kan nod 1 sända meddelandet "Hej" till nod 2 genom att bilda paketet

`[fas|fam|"Hej"]` (`fas`=fys. adress sändaren = nod 1, `fam`=fys. adress mottagaren = nod 2.)

Detta paket kablas ut och nod 2 hör sin adress (`fam`) nämnas. Digitaltekniken på nätverkskortet ordnar detta. Då vet nod 2 att meddelandet är ämnat för denne. Då lyssnar nod 2 och uppfattar `"Hej"`. Observera här hur den fysiska förbindelsen MÖJLIGGÖR kommunikation på den lokala nätverksnivån och att korten på detta sätt samarbetar och MÖJLIGGÖR kommunikation för operativsystemet som styr korten, operativsystemet får in meddelandet i buffrar och kan således behandla meddelandena vidare.

2.2 Mjukvarumässiga adresser: IP-adresserna

Vi har ovan beskrivit hur kommunikation går till över ett så kallat lokalt nätverk, *LAN = Local Area Network*. Den grundar sig som sagt på de fysiska adresserna som är inbrända på alla nätverkskort. En fysisk adress är då hårdvarumässig, den kan inte ändras (med mindre än att man bränner om innehållet i nätverkskortet.)

För att uppnå flexibilitet inför man en annan typ av adresser som kan ändras, det är de som är de så kallade IP-adresserna. Varje nätverkskort får tilldelat sig ett nummer (som kan ändras) uppbyggt av fyra bytes X.X.X.X (som vi vet) och då har ett lokalt nätverk. Vi ska studera en enkel typ av lokala nätverk. Inom denna typ ligger alla adresser inom ett intervall som beskrivs så här:

192.168.0.0 – 192.168.0.255

Det här är adresser till 256 noder. Observera hur de första tre bytesena är konstanta och det är endast den sista byten som varierar mellan noderna i detta nätverk. Man reserverar den första adresser (som slutar på 0) och den sista adresser (som slutar på 255) för speciella funktioner i detta lokala nätverk, så de adresser som är tillgängliga för noder i nätet är egentligen

192.168.0.1 – 192.168.0.254

men det gäller fortfarande att de första tre bytesena är konstanta.

De tre första bytesena är konstanta som sagt och man brukar då kalla adressen 192.168.0.0 för hela nätverkets adress, *network address*. Vidare reserveras den sista adressen till att sända ett meddelande till alla noder på nätet, det kallas *broadcast*. Den är här 192.168.0.255. Vidare finns det en teknisk term här som heter nätmasken, network mask, och den är 255.255.255.0 där vi sätter 255 på de bytes som är konstanta och en nolla på den delen av adressen som varierar med olika noder i nätverket.

Ett annat nätverk med andra adresser är

Network address: 192.168.1.0
Netmask: 255.255.255.0
Broadcast: 192.168.1.255

som då anger nätverket med noder i intervallet 192.168.1.1 – 192.168.1.254.

På nivå 2, där vi fortfarande befinner oss, har vi alltså en paketförmedlingstjänst mellan noder och en nod kan skicka "Hej" till en annan nod och dessa noder ligger in om ett adressintervall av den typ som vi just sett. Nu är det förstås inte "Hej" man skickar, man skickar faktiskt så kallade IP-paket, från en nod till en annan, och så länge IP-adresserna är inom samma intervall så lämnar vi inte det lokala nätverk som vi skickar inom. Men när en IP-adress hör till ett annat lokalt nätverk så måste det paket kunna lämna de lokala nätverket som det är skickat från. För att möjliggöra detta inför man det fiffiga, det briljanta det vill säga att man kopplar ihop nätverk av dessa typer med varandra. Man inför då en så kallad router som sköter paketförmedling mellan lokala nätverk, den har ofta adressen som slutar på 1 inom varje nätverk och en router är då alltså en nod som ingår i flera lokala nätverk. Det är det här som är inkörsporten till nästa nivå. En mer komplett specifikation över ett lokalt nätverk som är anpassat för att kommunicera med andra lokala nätverk

kan då alltså lyda så här:

Network address: 192.168.1.0
Netmask: 255.255.255.0
Broadcast: 192.168.1.255
GateWay: 192.168.1.1

De adresser som då alltså är tillgängliga för noder inom det lokala nätverket blir då alltså 192.168.1.2 – 192.168.1.254. Det paket som man bildar inom ett lokalt nätverk kan kallas lokalt nätverkspaket, den vanligaste typen är *ethernet*-paket som alltså har utseendet `[fas|fam|XXX]` där `fas` är den fysiska adressen hos sändaren, `fam` är den fysiska adressen hos mottagaren och `XXX` är det som man vill sända mellan de båda noderna.

3. Nätverksnivån. Denna nivå kopplar ihop de lokala nätverken. Alla noder inom ett lokalt nätverk kan som sagt nå varandra direkt genom att bara skicka ut ett paket, det uppfattas direkt av det nätverkskortet dit paketet ska. Men om alla noder i hela världen var i ett lokalt nätverk skulle det nätet bli väldigt belastat. Därför väljer man att införa så kallad internetteknik, (observera litet i!) och koppla ihop flera lokala nätverk till ett nät av nätverk, ett så kallat internet. Då måste ett paket korsa en *Gateway* för att ta sig över till ett annat lokalt nätverk. Vi går igenom detta steg för steg och vi antar att vi ska koppla ihop nätverken 192.168.1.0 och 192.168.2.0 med en gateway som har adresserna 192.168.1.1 samt 192.168.2.1. Båda nätverken har nätmaskerna 255.255.255.0 och nodernas adresser ligger alltså i intervallet 192.168.Y.1 till 192.168.Y.254 där Y är 1 eller 2. Gatewayen är då nod i båda de lokala nätverken med den adress som slutar på 1. Vi studerar hur ett paket skickas från nod 192.168.1.10 till 192.168.1.11 (alltså lokalt inom samma nätverk) samt hur ett paket skickas från 192.168.1.10 till 192.168.2.11 alltså över till det andra nätverket.

Fall 1: Från 192.168.1.10 till 192.168.1.11.

192.168.1.10 vill alltså skicka ett paket till 192.168.1.11 med innehållet XXX. Då bildar den ett IP-paket med följande principiella innehåll: `[ 192.168.1.10 | 192.168.1.11 | XXX ]`. Eftersom de båda noderna ligger i samma nätverk (192.168.1.0) behöver detta paket inte gå till gatewayen utan kan gå direkt via det lokala nätverket till sin adressat, 192.168.1.11. Hela det här IP-paketet förpackas då som innehåll i ett lokalt nätverkspaket med formen

```
[fas|fam|[ 192.168.1.10 | 192.168.1.11 | XXX ]]
```

där `fas` är den fysiska adressen hörande till noden med IP-adress `192.168.1.10` och `fam` är den fysiska adressen till noden med IP-adress `192.168.1.11`. Detta paket kablas ut och fångas upp av mottagaren och överföringen är fullbordad.

Fall 2: Från 192.168.1.10 till 192.168.2.11

192.168.1.10 vill alltså skicka ett paket till 192.168.1.11 med innehållet XXX. Då bildar den ett IP-paket med följande principiella innehåll: `[ 192.168.1.10 | 192.168.2.11 | XXX ]`. Detta paket skickas sedan ut på det lokala nätverket men eftersom mottagarnoden inte ligger i samma lokala nätverk så förpackas detta IP-paket i ett lokalt nätverkspaket med gatewayen som adressat istället, det vill säga, 192.168.1.10 bildar paketet

`[fas|fag|[ 192.168.1.10 | 192.168.2.11 | XXX ]]`

där `fas` är den fysiska adressen hörande till noden med IP-adress `192.168.1.10` och `fag` är den fysiska adressen till noden med IP-adress `192.168.1.1` som är detta lokala nätverks gateway. I gatewayen sitter en maskin som kallas *router* (engelska för vägväljare), routern kopplar ihop de två nätverken genom att förmedla paket till rätt nätverk. Routern öppnar nätverkspaketet och hittar IP-paketet `[ 192.168.1.10 | 192.168.2.11 | XXX ]`. Routern vet att mottagaren (`192.168.2.11`) finns på ett annat lokalt nätverk som också routern ingår i och routern bildar därför ett nytt lokalt nätverkspaket för det nätverket, det ser då ut så här:

`[fag|fam|[ 192.168.1.10 | 192.168.2.11 | XXX ]].`

Här är `fag` är den fysiska adressen hörande till noden med IP-adress `192.168.2.1` som är routern själv, i det lokala nätverket som mottagaren befinner sig i och `fam` är den fysiska adressen till noden med IP-adress `192.168.2.11`. När routern kablar ut detta paket på det lokala nätverket som 192.168.2.11 befinner sig i så är paketförmedlingen fullbordad då `192.168.2.11` fångar upp det.

Lägg märke till hur denna mekanism beskriver en paketförmedlingstjänst som levereras en nivå upp. Ovan så skickade vi "Hej" mellan två noder i ett lokalt nätverk, det är förstås inte bokstavligen "Hej" vi skickar, vi skickar IP-paket mellan noder i ett nätverk. Paketförmedlingstjänsten i de lokala nätverken möjliggör alltså att skicka IP-paket och det är detta som är IP-protkollet. Men vad innehåller ett IP-paket då? Jo, två IP-adresser (sändare och mottagare) och ett *innehåll* som vi markerat med `XXX`, vad är det då? Jo, det är något som i sin tur innehåller adresser och portnummer och andra uppgifter för att möjliggöra kommunikation av funktioner som ligger ännu högre upp. IP-protkollet möjliggör kommunikation mellan högre liggande funktioner. Och nästa lager i hierarkin kallas *Transportlagret* och här finner vi TCP och UDP.

Det viktiga att inse här är att den lagrade arkitekturen möjliggör en *problemuppdelning*. Vi skisserar detta genom att sammanfatta det vi gått igenom ovan:

1. Fysiska lagret tar hand om de fysiska problemen: hur stora ska spänningarna vara i signalkablarna, hur ser en kontakt ut?

2. Lokala nätverkslagret (tex *ethernet*) tillhandahåller en basal paketförmedlingstjänst inom ett lokalt nätverk baserat på de fysiska adresserna som är inbrända på nätverkskorten. Alltså hur ser signalföljden ut i signalkablarna mellan nätverkskorten och hur bildas ett lokalt nätverkspaket?

3. Nätverkslagret (IP) tillhandahåller en paketförmedlingstjänst mellan olika förbundna lokala nätverk och möjliggör också mjukvarumässiga adresser. Här finns som sagt IP-protokollet: hur ser ett IP-paket ut?

4. Transportlagret tillhandahåller UDP och TCP som är två varianter på hur man kan hantera en förbindelse mellan två IP-noder. UDP är en icke-felkontrollerande paketöverföringstjänst. Inom UDP har man heller inte en varaktig anslutning utan paketen till en viss försändelse skickas enskilt och det finns ingen felkontroll på att alla paketen kommit fram. Inom TCP har vi däremot felkontroll och en TCP-förbindelse är dessutom varaktig, den öppnas sedan sänder man och sedan stängs den.

2.2 Utseende i *Gentoo Linux* (och *UNIX* i allmänhet.)

Med kommandot `ifconfig` kan man se vilka inställningar som för närvarande råder i ett körande *UNIX/Linux*-system. Man måste vara `root` för att köra `ifconfig`. Vi tar en utskrift:

```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:26:b9:a3:0f:6e
          inet addr:192.168.0.2  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:22 Memory:f6ae0000-f6b00000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:56 errors:0 dropped:0 overruns:0 frame:0
          TX packets:56 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3336 (3.2 KiB)  TX bytes:3336 (3.2 KiB)
```

Här ser vi alltså att systemet för närvarande är anslutet till ett lokalt nätverk med adressen 192.168.0.0. Nätverkskortet med enhetsbeteckning `eth0` har IP-adressen 192.168.0.2 och kan således kommunicera över nätet. Namnet "`ifconfig`" är en förkortning för *interface configuration*, alltså gränssnittskonfiguration. I *Windows* heter motsvarande kommando `ipconfig` vilket alltså syftar på IP-konfigurationen, vilket är lite vilseledande, IP-konfigurationen är ju ett mer vidare begrepp, när man skriver **`if`**`config` eller **`ip`**`config` är man ju mer intresserad av ett *visst nätverkskort*, alltså ett *visst* ***interface***, så *UNIX*-namnet är mer genomtänkt än *Windows*-namnet.

3. Sockets (BSD-sockets = Internet domain sockets)

En *socket* är en av datorteknikens mest betydelsefulla uppfinningar. Ordet *socket* är engelska och betyder *anslutningsdon*, eller *uttag*. *Anslutningspunkt* skulle vara ett bra ord för att beskriva det och för att beskriva det i systemprogrammeringstekniska termer så är en socket en *fildeskriptor* som leder till en kommunikationskanal som är upprättad via ett IPC-protokoll. Vi har då tre olika typer av sockets:

1. Stream-sockets: bygger på TCP-protokollet, har alltså en varaktig anslutning med felkontroll.
2. Datagram-sockets: bygger på UDP-protokollet, har alltså ingen varaktig anslutning och ingen felkontroll.
3. Raw-sockets: bygger direkt på IP-protokollet, ger oss alltså möjlighet att skicka IP-paket utan att gå via TCP- eller UDP-protokollen.

Vi ska inte fördjupa oss så mycket i dessa olika varianter, vi ska bara använda TCP-sockets och poängtera IPC-aspekten här, en *socket* är som sagt en *fildeskriptor* som upprättas som en kommunikationskanal som stödjer sig på de tidigare protokollen som vi gått igenom. Ett annat exempel på en fildeskriptor är ju en *pipe* som också möjliggör kommunikation mellan två processer, med kommunikation med pipes kommunicerar man ju mellan två processer på samma dator medan sockets alltså möjliggör kommunikation mellan två processer som kan befinna sig på två olika datorer om dessa datorer är förbundna över IP.

Sockets uppfanns som del av systemet *BSD-UNIX*, en av de klassiska *UNIX*-varianterna, och är som sagt så pass lyckad att man har infört den i de flesta *UNIX*-system. Vi ska ni se närmare på hur det ser ut systemprogrammeringsmässigt i *C*.

3.1 Sockets i *C (Internet domain)*

Vi kommer att basera detta avsnitt på *Beej's Guide to Network Programming* som finns på nätet. *Beej's guide* är kurslitteratur som ni alltså måste läsa ordentligt. Den lämpar sig mycket väl för självstudier, så jag kommer endast översiktligt gå igenom en del saker här.

Den *C*-kod som vi använder ser ut på följande sätt. Klientprogrammet `client.c`:

```
/*client.c -- a stream socket client demo*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3490 // the port client will be connecting to
#define MAXDATASIZE 100 // max number of bytes we can get at once

int main(int argc, char *argv[])
{
      int sockfd, numbytes;
      char buf[MAXDATASIZE];
      struct hostent *he;
      struct sockaddr_in their_addr; // connector's address information

      if (argc != 2) {
          fprintf(stderr,"usage: client hostname\n");
          exit(1);
      }

      if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
          perror("gethostbyname");
          exit(1);
      }

      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
          perror("socket");
          exit(1);
      }


      their_addr.sin_family = AF_INET;    // host byte order
      their_addr.sin_port = htons(PORT);  // short, network byte order
      their_addr.sin_addr = *((struct in_addr *)he->h_addr);
      memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);


      if (connect(sockfd,(struct sockaddr*)&their_addr,sizeof their_addr)==-1) {
          perror("connect");
          exit(1);
      }
```

```
      if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
          perror("recv");
          exit(1);
      }

      buf[numbytes] = '\0';
      printf("Received: %s",buf);
      close(sockfd);
      return 0;
}
```

Vi ser på motsvarande program på serversidan som ska ta emot anslutningar av ovanstående klientprogram, `server.c`:

```
/*
** server.c -- a stream socket server demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
#define MYPORT 3490     // the port users will be connecting to
#define BACKLOG 10 // how many pending connections queue will hold

void sigchld_handler(int s)
{
      while(waitpid(-1, NULL, WNOHANG) > 0);
}

int main(void) {
      int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
      struct sockaddr_in my_addr;   // my address information
      struct sockaddr_in their_addr; // connector's address information
      socklen_t sin_size;
      struct sigaction sa;
      int yes=1;

      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
      }

      if (setsockopt(sockfd,SOL_SOCKET, SO_REUSEADDR, &yes,sizeof(int)) == -1) {
            perror("setsockopt");
            exit(1);
      }

      my_addr.sin_family = AF_INET;         // host byte order
      my_addr.sin_port = htons(MYPORT);     // short, network byte order
      my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
      memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);
```

```
	if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr) == -1) {
		perror("bind");
		exit(1);
	}

	if (listen(sockfd, BACKLOG) == -1) {
		perror("listen");
		exit(1);
	}

	sa.sa_handler = sigchld_handler; // reap all dead processes
	sigemptyset(&sa.sa_mask);
	sa.sa_flags = SA_RESTART;
	if (sigaction(SIGCHLD, &sa, NULL) == -1) {
		perror("sigaction");
		exit(1);
	}

	while(1) {  // main accept() loop
		sin_size = sizeof their_addr;
		if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
		                                          &sin_size)) == -1) {
			perror("accept");
			continue;
		}
		printf("server: got connection from %s\n",
		                          inet_ntoa(their_addr.sin_addr));
		if (!fork()) { // this is the child process
			close(sockfd); // child doesn't need the listener
			if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
				perror("send");
			close(new_fd);
			exit(0);
		}
		close(new_fd);  // parent doesn't need this
	}
	return 0;
}
```

Detta är den mest basala formen av ”*Client-Server*”, det är er uppgift att ta dessa program och lägga in dem i laboration 2 och också ta demon/server-funktionaliteten från laboration 1. Client-server betyder att de olika kommunicerande processerna i de båda ändarna har *olika* status, den ena är en *server* (som kan acceptera anrop av flera klienter samtidigt) och den andra är en *klient* som bara kan anropa en server i taget. Man kan dock köra igång flera instanser av klientprogrammet och låta det ansluta till en och samma server. Observera att det då är en dialog per klientanslutning som förekommer. Detta möjliggörs av `accept()` som delegerar själva kommunikationen med den anropande klienten till en barnprocess i serverprogrammet. Med denna teknik/konstellation kan servern, som gör `accept()`, acceptera och upprätthålla flera *parallella* dialoger med olika anslutande klienter. Att en process delegerar kommunikationen till en underprocess eller ny tråd är en förutsättning för *Client-Server*, istället för *Peer-to-Peer* (eng. *peer* = like). Det är ofta för att servern tillhandahåller någon form av tjänst, själva den kommunikation som utgör själva tjänstens konkreta tillhandahållande sker då i en barnprocess eller ny tråd.

3.2. Plattform

Om ni har genomfört installationen av ett operativsystem så har ni två *UNIX*-system som använder BSD-sockets, dels *Gentoo Linux* eller *Arch Linux* och dels *Debian*. Då inga system kör, ställ in

VirtualBox på att inte dela ut IP-adresser dynamiskt utan välj läget "*attached to internal network*" i *Network*-inställningarna. Tilldela dessa sedan två system statiska IP-adresser (med `ifconfig`), till exempel 192.168.1.1 samt 192.168.1.2. (Detta kan göras för hand som `root` när systemen kör.) Sedan kan du kontrollera att de båda virtuella maskinerna man nå varandra med `ping` och sedan kan du börja genomföra nätverksprogrammeringen. Troligtvis måste dina program kompileras på följande sätt:

```
> gcc program.c -o program -lnsl
```

Man kan även behöva `-lsocket` och `-lresolv` som det står i Beej's kompendium. Ni får pröva er fram vilka bibliotek som behövs. Dagens övning går ut på att få igång de båda testprogrammen `client.c` och `server.c`, det ena på *Gentoo/Arch*-systemet och det andra på *Debian*-systemet.

3.3. Typer av sockets

Det kan vara bra att läsa om de olika typerna av sockets (stream, datagram och raw) antingen i Beej's Guide, eller kanske `www.wikipedia.org`. Ni kommer absolut att behöva ha förståelse för stream- och datagramsockets i programvaruprojektet senare i vår. Då kommer ni dock även att ha fått en ordentlig genomgång av nätverksteori och teknik i den delkurs i programvaruprojektet som hetet Cisco-delen. Den lär också, vad jag vet, kunna leda till någon form av Cisco-certifiering och det är bra. Cisco är ju en av världens största aktörer inom nätverksteknik. I operativsystemskursen ska vi främst se på de systemprogrammeringstekniska aspekterna av detta, men det kan vara bra att börja grubbla över deras sammanghang redan nu.

Hur man skapar ett lokalt nätverk mellan virtuella maskiner med hjälp av VirtualBox:

1. ***Välj ut de virtuella maskiner som ska ingå i det lokala nätverket.***
2. ***Gå till inställningarna och leta upp fliken om nätverkskortet "Network adapter", välj där attached to: "internal network".***
3. ***Ändra i de virtuella maskinerna så att de har statiska IP-adresser och välj 192.168.0.1 för en av dem, 192.168.0.2 för nästa och så vidare.***
4. ***Starta maskinerna och gör ping 192.168.0.1 från den som har Ip 192.168.0.2 och tvärtom för att se att de har kontakt. Om de har kontakt så fungerar det interna nätverket.***

Hur man ansluter ett lokalt nätverk till Internet (överkurs, jag har heller inte testat detta):

1. ***Skapa en minimal installation av systemet Debian, utan grafiskt användargränssnitt.***
2. ***Skapa två nätverkskort, kalla dem eth0 och eth1. Anslut den ena till det lokala nätverk du vill ansluta till Internet och anslut det andra via NAT.***
3. ***Googla på "How to enable IP Forwarding in Debian" och utför denna manöver.***

Som sagt, jag har inte provat detta i en virtuell miljö men principiellt bör det fungera. (Famous last words.) Den minimala debianmaskinen kommer att fungera som router och det ena nätverkskortet behöver ha en statisk IP-adress och det andra (som är anslutet till nätet via NAT) behöver få IP via DHCP.