Femte Generationens Operativsystem

Fredrik Tolf
<fredrik@dolda2000.com>

22 november 2004

Sammanfattning

I ett års tid har jag utformat planer på hur man bör bygga ett operativsystem, som jag anser är bättre än det allra mesta som används och byggs i dagsläget. Då planerna nu är så långt gångna att de faktiskt är redo att implementeras i praktiken, ämnar jag beskriva detta system samt argumentera för varför jag anser det vara bättre än det allra mesta som används och byggs i dagsläget. Texten kommer också visa varför jag anser detta system värdigt att kallas ett ”nästa generationens” system.

Viktiga punkter i genomgången innefattar t.ex. varför systemet bör bygga på en mikrokernel, hur den grundläggande exekveringsmiljön ser ut, vilka grundläggande tjänster systemet tillhandahåller, samt varför och hur man ska åstadkomma systemtransparens och ett system som är lika globalt som Internet.

Innehåll

1 Inledning

1.1 Bakgrund

För ungefär ett år sedan trodde jag att anledningen till att jag inte tycker om C++ var därför att de operativsystem som program skrivna i C++ körs på inte är objektorienterade, och att man därför inte bör försöka skriva objektorienterade program till icke objektorienterade operativsystem. Därför började jag klura på hur ett objektorienterat operativsystem, som ett språk som C++ skulle kunna passa till, borde se ut.

Under mina funderingar upptäckte jag dock snabbt att operativsystemets arkitektur trots allt inte var anledningen till att jag inte tycker om C++. Jag hade dock kommit på tillräckligt många intressanta idéer om hur ett operativsystem borde byggas att jag inte bara kunde slänga bort dem av den anledningen. Jag fortsatte därför att bygga vidare på dessa idéer, dock utan objektorienterade vinklingar. I sinom tid utvecklade jag teorierna långt nog att jag skulle vilja påstå att de kan användas som grundval för att implementera ett operativsystem, som skulle lösa många av de kvalitativa problem som dagens operativsystem har. Den här artikeln är ett försök att presentera dessa idéer, samt att argumentera för varför de är ”bättre” än dagens populära operativsystem.

I boken ”Modern Operating Systems” [1] delar professor Andrew Tanenbaum in operativsystem i fyra generationer – en kort sammanfattning följer:

1. 1:a generationens operativsystem är i princip en dator utan operativsystem. Användarna bokar tid på datorn och får den alldeles för sig själva ett antal timmar.

2. Ett 2:a generationens operativsystem är ett batchsystem, som kan läsa in program automatiskt och exekvera dem sekventiellt.

3. 3:e generationens operativsystem inkluderar främst sådana funktioner som multiprogrammering och time sharing, d.v.s. möjliggör att flera personer använder datorn samtidigt.

4. Ett 4:e generationens operativsystem är antingen ett nätverksoperativsystem – i princip ett 3:e generationens system som har fått lite nätverkstjänster påskruvade i efterhand – eller ett distribuerat system, i vilket flera fysiska datorer samarbetar för att se ut som ett system i användarens (och helst även programmerarens) ögon.

Jag anser att det operativsystem jag ämnar redogöra för innehåller kvalitativa skillnader, vilka gör det svårt att klassificera det som något av ovanstående, och vill därför klassificera det som ett 5:e generationens operativsystem – därav titeln.

1.2 Språkbruk och terminologi

1.2.1 Operativsystem

Det finns flera definitioner av ordet operativsystem i bruk. Professor Tanenbaum definierar det i [1] som den del av systemet som har direkt tillgång till all hårdvara och sköter resursallokering – d.v.s. det program man normalt bemärker med det engelska ordet *kernel*.

Jag och många andra är dock inte av den uppfattningen. Genom den här rapporten tänker jag använda ordet operativsystem för att syfta på den samling av program som utgör den grund som andra program behöver för att köra, och som användare behöver för att kunna använda en dator. Det finns naturligtvis många alternativ för vad detta kan tänkas innefatta, men vanligt är en kernel, ett antal standardbibliotek med subrutiner som program kan använda, en kompilator, ett användargränssnitt, samt en liten uppsättning program som kan anropas genom användargränssnittet för att utföra vissa vanliga sysslor.

1.2.2 Kernel eller kärna?

I rapporten kommer jag att använda ordet *kernel* för att syfta på det program som har direkt tillgång till hårdvaran i datorn. Vissa hävdar att man bör översätta det till *kärna* på svenska. För det första hävdar jag att det blir konstigt om man gör det, då *kernel* har en viss idiomatisk betydelse i datorterminologi som inte innehas av ordet *kärna*. För det andra, och än viktigare, kan jag inte använda *kärna* för det syftet i den här rapporten, då jag har ett annat begrepp, som jag på engelska kallar *core*, vilket jag hellre vill översätta till *kärna*.

Jag konstruerar ordet *kernelen* för den bestämda formen av *kernel*, *kernelar* för den obestämda pluralformen, samt *kernelarna* för den bestämda pluralformen.

1.2.3 Monolitisk kernel samt mikrokernel

Inom operativsystemsarkitektur brukar man skilja mellan en *monolitisk kernel* och en *mikrokernel*. En monolitisk kernel är en kernel som, förutom grundläggande hårdvarukontroll och resursallokering även sköter vissa bekvämlighetsfunktioner för användarprogram, t.ex. filsystem och nätverksfunktionalitet.

En mikrokernel, å andra sidan, innehåller endast de mest grundläggande funktionerna som en kernel måste ha, och låter användarprogram ta hand om de ovan nämnda bekvämlighetsfunktionerna. Vanligtvis har man, i ett mikrokernel-baserat operativsystem, ett program som sköter om filsystemet, och låter användarprogram kommunicera med detta program genom kernelen.

1.2.4 Paging och PMMU

PMMU står för *Paging Memory Management Unit* eller *Paged Memory Management Unit*, beroende på vem man frågar. Innebörden är dock densamma i båda fallen. När ett program förr i världen efterfrågade att läsa en byte ur minnet på den datorn det kördes på, läste processorn exakt från den adress i minnet som programmet efterfrågade. Två konsekvenser av detta var säkerhetsproblem (om två program körde samtidigt på datorn fanns inget som förhindrade det ena att läsa och modifiera det andras minne) samt att det programmet som kördes inte kunde vara större än det fysiska minnet i datorn.

Det första av dessa problem fick en temporär lösning i och med segmentering vilken jag inte ämnar redogöra vidare för, eftersom det är uråldrig historia vid det här laget. Önskar läsaren information om det, finns det i [1]. För att lösa det andra problemet uppfanns *Paging* – jag kommer fortsätta använda den engelska termen helt enkelt för att jag inte känner till någon tillfredsställande översättning; om det finns någon, var god upplys mig om den så byter jag genast ut alla förekomster av ordet – tack vare vilket det blev möjligt att köra program som tog upp mer minne än man hade rent fysiskt.

Paging går ut på att man har en extra hårdvaruenhet mellan processorn och minnet (denna sitter nuförtiden oftast på samma chip som processorn själv, men det är irrelevant), nämligen en PMMU. När processorn vill läsa från eller skriva till minnet går adressen genom PMMU:n, vilken har möjligheten att ”skriva om” adressen innan den når minnet. Detta tar formen att PMMU:n tittar på de översta bitarna i adressen och byter ut dem mot andra bitar genom att titta i en översättningstabell som kernelen har skapat (denna tabell kallas för *Page Table*, eller sidtabell).

Det riktigt intressanta, utöver själva sidöversättningen, är att kernelen dessutom kan sätta en bit eller två i sidtabellen som avgör om sidan finns tillgänglig eller inte. Om ett program försöker läsa från eller skriva till en sida i minnet vars bitar inte är på, kommer PMMU:n signallera till processorn att kernelen får ta över.

Från början användes detta, som sagt, till att låta program vara större än det fysiska minnet i datorn: om det inte fanns tillräckligt minne ledigt, kunde kernelen välja ett antal sidor och skriva dem till disk, sätta åtkomstbitarna i sidtabellen till 0, och använda de frigjorda sidorna till annat. När det program, vars sidor kernelen tog, återigen försökte komma åt de sidorna, lät PMMU:n kernelen ta över och läsa tillbaka sidorna från disken (eventuellt genom att ta sidor från ännu något annat program), för att sedan låta programmet fortsätta som om inget hade hänt.

Detta är naturligtvis fortfarande ett populärt användningsområde för paging, men man upptäckte snabbt att det dessutom hade många andra intressanta användningsområden. Man kan t.ex. genom sidöversättning låta

två program dela på samma minnessidor på ett säkert och kontrollerat sätt.

För mer information om paging, se [1].

1.2.5 ”Boota”

Även om det engelska ordet *boot* (från *bootstrap*) mycket väl kan översättas med *starta* på svenska, anser jag att *boot* har en så pass idiomatisk betydelse just i och med att det syftar till att starta igång ett operativsystem, och ingenting annat, att man bör använda det för att vara tydlig på punkten att det är just ett operativsystem som startas, och på så vis undvika missförstånd.

1.2.6 Kernel- och användarläge

Nästan alla processorer nu för tiden har två exekveringslägen: *kernelläge* samt *användarläge*. Ett program som kör medan processorn är i kernelläge har fullmakt över all hårdvara i hela datorn (detta program är allt som oftast kernelen – därav namnet). Ett program som kör medan processorn är i användarläge har bara tillgång till de resurser som kernelen har delat ut till det programmet. Ett sådant program kallas oftast *användarprogram*. Ofta behöver man dock även en term som syftar till ett program som körs p.g.a. att det åberopats av en användare, i kontrast till ett program som körs p.g.a. att systemet åberopat det. I brist på termen användarprogram för att syfta på ett sådant program, använder jag istället *användareprogram* (notera tillägget av ett *e*).

1.2.7 Systemanrop

Ett *systemanrop* är en speciell sorts funktionsanrop, som ett användarprogram kan använda för att anropa en funktion som körs i kernelläge. Oftast har processorn en speciell operation just för utföra systemanrop. När denna operation utförs, försätter processorn sig själv i kernelläge och kör en funktion som kernelen tidigare har ställt in processorn på att köra när situationen uppstår.

1.3 Nuvarande problem

Idag används ett antal populära operativsystem. På persondatorer härskar Microsoft® Windows® suveränt. Även om det finns uppstickare såsom olika BSD-varianter, Linux™ och Mac OS X, kommer dessa system inte ens tillsammans upp till mer än en bråkdel av Windows® marknadsandel i dagsläget. På större datorer finns det större heterogenitet, men trots att det är i andra proportioner, är det ungefär samma systemtyper man träffar på. Även de allra största datorerna kör allt som oftast ett Unix-liknande system.

Samtliga system nämnda ovan är s.k. *nätverksoperativsystem*, vilket innebär att varje dator som kör systemet kan kommunicera med andra datorer som antingen kör samma system, eller ett annat system som använder samma kommunikationsprotokoll.

Den senaste tiden har många av de ovan nämnda systemen börjat implementera olika klustringstekniker, med hjälp av vilka man kan få flera datorer, som kör samma system, att samarbeta och se ut som en dator i både användarens och programmerarens ögon. Därför uppfyller de systemen kraven för att kunna kallas distribuerade system.

Då jag har använt datorer ganska länga, har jag märkt av många av de problem som normalt brukar märkas ut för de ovannämnda systemen. Jag tänker här göra ett försök att peka ut dem.

1.3.1 Flexibilitet

Samtliga ovan nämnda system är baserade på monolitiska kernelar. Detta innebär att kernelen ger program en viss implementation av vissa grundläggande tjänster, t.ex. filsystemet, och programmen har sedan inget annat val än att använda just det filsystemet. Förvisso kan ett program implementera sitt eget filsystem och använda det, men de flesta känner att det är onödigt och klumpigt när kernelen fakiskt ändå bidrar med ett operativsystem och tvingar sig att använda det i alla fall. I synnerhet finns inget sätt att förhindra ett program från att använda det system som kernelen tillhandahåller.

Vissa program har gått så långt som att faktiskt implementera att eget filsystem. Särskilt noterbart är det fria projektet GNOME, vars mål är att bygga en skrivbordsmiljö för Unix-liknande operativsystem, som del av GNU-projektet. Till GNOME hör ett funktionsbibliotek som kallas gnome-vfs, som tillhandahåller ett utökat filsystem, som bygger på URL:er. Man kan även komma åt det lokala filsystemet som kernelen tillhandahåller genom en `file://`-URL.

Det är dock inte bara filsystemet som lider av inflexibilitet på monolitiska kernelar. Vanliga subsystem som lider av samma problem, men som är betydligt svårare att implementera utanför kernelen, är t.ex. minneshantering, nätverksprotokoll och CPU-schemaläggning.

1.3.2 Systemgränser

I alla operativsystem som jag känner till är systemet en odelbar enhet, och tydligt avgränsad från andra system. På de ovan nämnda systemen är ett system begränsat till exakt en fysisk dator, och alltså är en dator avgränsad från andra datorer.

En dator kan förvisso kommunicera med andra datorer över t.ex. ett lokalt nätverk, men kommunikationen är sällan särskilt transparent för användaren. Den är oftast ännu mindre transparent för programmeraren. Som

exempel kan nämnas hur otransparent det är att köra ett program på en annan fysisk dator – om jag är inloggad på datorn `red01.nada.kth.se` och vill köra ett beräkningsintesivt program på datorn `my.nada.kth.se`, eftersom den sistnämnda har avsevärt mer kraftfulla processorer, måste jag manuellt logga in på `my.nada.kth.se` och köra programmet. Än värre är det faktum att, om jag har ett användarkonto på KTH, så kan jag inte använda det för att logga in på min dator hemma och komma åt mina filer på KTH – för att göra detta måste jag skapa mig en separat inloggning på KTH över något visst protokoll och utföra mitt arbete där. Likaså om jag skulle befinna mig på ett Internet-kafé utomlands, skulle jag inte kunna logga in på vare sig mitt konto hemma eller på KTH. (Vissa system tillåter det till en viss, om än liten, del – mer om detta i sektion 4)

Distribuerade system löser vissa av dessa problem – främst möjligheten att köra ett program på en mer kraftfull dator transparent – men bibehåller vissa andra. Alla system jag har studerat har haft det problemet att de har en oövervinnerlig administrativ gräns vid systemets kant – även om systemet spänner sig över flera datorer – som gör det omöjligt att komma till en dator var som helst i världen och logga in på sitt vanliga konto.

2 Exekveringsmiljö

Det system jag har utvecklat (om än bara i tanken, än så länge), tycks mig ha möjligheten att övervinna de ovan nämnda problemen. Som den reduktionist jag är vill jag gärna börja beskriva systemet från grunden och sedan förklara vill implikationer detta får. Därför börjar jag med den mest grundläggande delen av ett system: den exekveringsmiljö kernelen ger ett program att köra i.

2.1 Grundläggande arkitektur och systemkrav

Jag har valt att basera systemet på en mikrokernel, då jag anser det vara det enda sättet att övervinna flexibilitetsproblemet. Vidare har en mikrokernel i princip endast en praktisk nackdel, nämligen ren prestanda. Hur sant detta är har även det diskuterats – undersökningar [2] visar att de fördröjningar som skapas p.g.a. den extra kommunikationen som måste ske i ett mikrokernelbaserat system orsakar väldigt små prestandaförluster i förhållande till det arbete som ändå ska utföras.

Vidare har jag valt att kräva att systemet körs på en dator som har en PMMU. Detta förhindrar förvisso systemet från att köras på de allra minsta datorna, men de allra flesta processorer – till och med de flesta mobiltelefoner – har idag trots allt en PMMU, och jag anser att fördelarna överlägset väger upp nackdelarna.

2.2 Kärnor

Kernelen kommer att hantera ett antal objekt. Det mest grundläggande av dessa kallar jag en *kärna* (från engelskans *core* – ett ord som syftar på minne, från *ferrite core memory*). En kärna är främst en virtuell adresseringsrymd, alltså i princip en sidtabell. En kärnas främsta syfte är att ett program och dess dataminne kan mappas in i den med hjälp av PMMU:n. När systemet först bootas kommer bootladdaren att ha läst in ett par program från disken till minnet innan kernelen har fått kontroll, och när kernelen har initialiserat sig själv, konstruerar den de första kärnorna med hjälp av dessa förladdade program. De viktigaste programmen som kommer ha lästs in är följande:

- Den anonyma minnesmapparen
- Filsystemsservern
- Exec-servern
- Initieringsprogrammet
- Diverse drivrutiner, framförallt de för disk- och konsoll-styrning.

Varje kärna i kernelen tilldelas ett nummer för identifikation – detta nummer kallas CID (för Core ID), och representerar alltså unikt en kärna.

2.3 Trådar

Det näst mest grundläggande kernelobjektet är en *tråd* (från engelskans *thread*). En tråd representerar ett programs körbara status i en kärna, d.v.s. värdet på alla register i processorn, instruktionspekarens värde samt vilken kärna programmet körs i. En tråd är bunden till en viss kärna som den inte kan lämna.

I likhet med kärnor får varje tråd ett unikt nummer. I analogi med CID kallas detta för TID.

Det första kernelen gör efter att den har konstruerat de tidigare nämnda kärnorna är att den skapar en tråd som börjar köras på en bestämd adress i initieringsprogrammet.

2.4 Objektreferenser

Det är bara kärnor och trådar som har kernel-globala identifikationsnummer. Alla kernelobjekt har ett lokalt identifikationsnummer som går att nå från den kärnan i vilken de först skapades. Ett sådant nummer kallas OID (för Object ID). CID- och TID-nummer används endast sällan. Normalt är det kärnornas och trådarnas OID-nummer som används vid systemanrop.

När systemet just bootats lägger kernelen in OID-nummer i initieringsprogrammets kärna som syftar på portar (se sektion 2.5) till de förladdade kärnorna.

Kernelen kör med jämna mellanrum en skräpinsamlare (garbage collector), som märker ut de kernelobjekt till vilka det inte längre finns några referenser, och förstör dem för att få tillbaka minnet de använde.

Kernelen tillhandahåller tre systemanrop för referensmanipulering:

- PUT: Tillåter en tråd att indikera att den inte tänker använda en viss objektreferens längre. Det motsvarande OID-numret blir då ogiltigt för framtida användning i den kärnan det tidigare användes, och kernelen noterar att den motsvarande objektrelationen inte längre finns.

- DUP: Tar ett OID-nummer och skapar och returnerar ett nytt OID-nummer som syftar på samma objekt.

- WEAK: Tillåter en objektreferens att markeras som en ”svag” referens. Kernelen kan samla in objekt som bara refereras till med svaga referenser som skräp. Samma systemanrop tillåter referensen att avmarkeras som svag (med andra ord, återställas som en ”stark” referens).

2.5 Portar

Det primära kernelobjektet för användning inom inter-kärn-kommunikation kallas *port* (från engelskans *task gate* – förslag på bättre översättning skulle inte skada). En port motsvarar en viss adress i en kärna.

Kernelen tillhandahåller ett systemanrop, CALL, vilket tillåter en tråd i en kärna att ange en port via dess OID. När detta systemanrop utförs skapar kernelen en tråd i den kärnan som porten refererar till, på den adressen som porten anger och låter den köra (jag tar upp multitasking och schemaläggning senare – låt oss för tillfället bara anta att de två trådarna kommer köras samtidigt som i ett vanligt multitaskande system). Vid systemanropet tillåts även att den anropande tråden skickar med ett valfritt antal numeriska argument till den nya tråden, samt ett strängargument (detta är tänkt att representera en av flera möjliga funktioner som kan anropas över porten). Systemanropet returnerar ett OID-nummer som syftar på den nya tråden. Vidare tillhandahåller kernelen ännu ett systemanrop, WAIT, till vilket en tråds OID anges, som blockar den anropande tråden till dess att den angivna tråden returnerar. Detta görs genom ett tredje systemanrop, RET. Till RET skickar tråden även ett valfritt antal numeriska argument. Dessa argument returneras av WAIT till den tråd som väntar. På detta sätt skapas ett sätt för trådar att utföra RPC-anrop (Remote Procedure Call) i andra kärnor.

Det kommer att finnas mekanismer för att skicka objektreferenser till den nya tråden via CALL och tillbaka till den anropande tråden via RET. I båda fallen skickar den systemanropande tråden OID-nummer för de kernelobjekt den ämnar skicka till den motsvarande tråden, och kernelen skapar

nya OID-nummer i den kärnan i vilken den motsvarande tråden körs, så att den motsvarande tråden kan syfta till de medskickade kernelobjekten.

Kernelen tillhandahåller också ett systemanrop, CRGATE, med vilket ett program kan skapa en ny port. Tar som argument adressen vid vilken tråden ska börja köra när porten anropas, samt ett numeriskt argument som återges av kernelen när porten anropas. Det senare är tänkt att kunna användas som en identifikation för det anropade programmet, om flera portar anropar samma adress.

I och med att systemet bootas lägger kernelen även in ett antal portar i initieringsprogrammets kärna. Ett anrop på någon av dessa portar går direkt in i kernelen för att utföra olika skyddade funktioner (dessa kommer nämnas efterhand). Initieringsprogrammet kan sedan dela ut dessa portar till andra program som bör ha tillgång till dem genom RPC-anrop.

2.5.1 Fördelar med RPC

Låt mig gå in på en liten diskurs om varför jag har valt just RPC-anrop som den kernel-primitiva kommunikationsmetoden. Alternativen till RPC-anrop är i regel byte- eller datagram-orienterade strömmar (t.ex. TCP- eller SPX-liknande protokoll). Dessa tre är exakt lika vad gäller kommunikationsmöjligheter. Det är ett påstående som är enkelt att bevisa eftersom man ovanpå byte-strömmar kan designa ett protokoll som emulerar både datagram-strömmar och RPC-anrop (Sun RPC), på samma sätt som man ovanpå datagram-strömmar kan emulera byte-strömmar (TCP) och RPC-anrop (Sun RPC), och på samma sätt som man ovanpå RPC-anrop kan emulera både byte-strömmar och datagram-strömmar (båda illustreras av NFS).

Jag har dubbla anledningar att välja en RPC-implementation för kommunikation. Först och främst anser jag att det skapar ett mer uniformt och mindre komplext system, eftersom funktionsanrop redan är den mest primitiva kommunikationsmetoden inom ett program – på det viset blir skillnaden mellan primitiva funktionsanrop och RPC-anrop minimal, och kan i bästa fall göras helt transparent, vilket minskar komplexiteten för programmeraren. Den andra anledningen är att det enklare tillåter skapandet av ett mer händelsedrivet system, eftersom ett RPC-anrop i den här modellen skapar en ny tråd, istället för att ett program måste ha en tråd som ständigt lyssnar efter nya meddelanden från andra program i systemet. Det integrerar dessutom kommunikationen och skapandet av nya trådar i en primitiv funktion, vilket återigen minskar systemets komplexitet. För goda anledningar att minska ett systems komplexitet, se [3].

Det ska naturligtvis sägas att det finns anledningar att förespråka dataströmmar istället för RPC-anrop. Den främsta av dessa är att det minskar komplexiteten, eftersom processen att överföra data kräver mindre intern struktur än vad ett RPC-anrop gör. Även i jämförelse anser jag dock fort-

farande att RPC-anrop är lämpligast för den primitiva formen av kommunikation. Dels är det så, att, för att implementera en dataströms-orienterad kommunikationsmetod, krävs ett anrop (förmodligen till kernelen) för att initiera överföringen av data. Detta anrop kan lika gärna vara ett RPC-anrop som ett systemanrop, och därför finns inga som helst förhinder för de applikationer där dataströmskommunikation är lämpligast. För det andra är det så att, vid en snabb överblick av de vanligaste protokollen som används på Internet, ser man att de allra flesta faktiskt implementerar en sorts RPC-anrop över en dataströmskanal (t.ex. SMTP, POP3, IMAP, DNS, FTP, Kerberos, NNTP, NFS och NIS, bara för att nämna ett fåtal). I ljuset av detta verkar det mest statistiskt riktiga att implementera den primitiva kommunikationskanalen med hjälp av RPC-anrop.

2.6 Mappare

För att bygga upp andra kärnor än de som skapas av kernelen och bootladdaren, används *mappare* (av engelskans *mapper* – förslag på bättre översättning tages tacksamt emot). En mappare motsvaras av två separata kernel-objekt: ett styrobjekt och ett klientobjekt. När ett program skapar en mappare returnerar kernelen båda dessa. Klientobjektet är sedan avsett att överföras till en annan kärna via ett portanrop.

Via styrobjektet och ett systemanrop kan ett program ”lägga in” ett antal minnessidor i mapparen. Ett annat program kan sedan, via klientobjektet och ett annat systemanrop, mappa in dessa sidor (eller en del av dem) i den kärna, i vilken programmet körs. Via styrobjektet kan sedan tillgänglighets- och skrivskydds-bitarna sättas individuellt för varje mappning. När mapparen skapas skickar det skapande programmet även en port till kernelen, vilken sedan kommer anropas från kernelen när ett program som har mappat sidorna i mapparen råkar ut för ett sidfel p.g.a. att det försöker komma åt en sida vars tillgänglighetsbitar inte är på, så att programmet som skapade mapparen kan åtgärda detta. Tråden som utförde sidfelet blockas automatiskt av kernelen till dess att den nya tråden i den styrande programmet har returnerat, och startas då om på samma ställe som sidfelet inträffade på.

För att få tomma minnessidor till att börja med, utnyttjas en av de administrativa portar som initieringsprogrammet får i och med att systemet bootas. Ett anrop till denna port returnerar ett klientobjekt till en mappare av rena fysiska sidor utan bieffekter. Eftersom dylika minnessidor är en högst begränsad tillgång, kommer endast vissa privilegierade program att få tillgång till den porten, i synnerhet den anonyma mapparen. Den anonyma mapparen är, som ovan nämnt, ett av de program som laddas av bootladdaren innan kernelen startas. Dess uppgift är att dela ut anonymt minne, d.v.s. minne som kan skrivas ut till disk om systemet inte har tillräckligt många lediga fysiska sidor.

För mappare finns sju systemanrop:

- CRMAP: Skapar en ny mappare. Tar som argument adressen för den första sidan i mapparen, antalet sidor som ska ingå, samt ett OID-nummer som refererar till porten som ska anropas vid sidfel.

- MODMAP: Modifierar åtkomstbitarna för en sida i mapparen. Tar som argument ett OID-nummer som refererar till styrobjektet för mapparen, ett sidnummer, samt de nya åtkomstbitarna.

- MAPPAGE: Byter ut en sida i en mappare mot en annan sida. Tar som argument ett OID-nummer som refererar till mapparens styrobjekt, sidnumret i mapparen samt adressen till den nya sidan.

- MAPREF: Returnerar och återställer referensbitarna för en sida. Tar som argument ett OID-nummer som refererar till en mappares styrobjekt samt ett sidnummer.

- MAPLEN: Returnerar antalet sidor i en mappare. Tar som argument ett OID-nummer som refererar till en mapparen klientobjekt.

- MAP: Mappar in ett antal sidor från en mappare i den lokala kärnan. Tar som argument ett OID-nummer som refererar till en mappares klientobjekt, första sidan som ska mappas in, samt antalet sidor som ska mappas in.

- UNMAP: Tar bort en tidigare mappning. Tar som argument adressen till första sidan i mappningen.

2.7 Minneshål

Det krävs ett sätt för program att skicka icke-numeriska argument över RPC-anrop. Det främsta exemplet är strängar och datastrukturer. Detta problem skulle kunna lösas genom att det anropande programmet allokerar en tom sida, lägger strängen/datastrukturen i början av sidan, skaparen en mappare för den sidan och skickar över ett klientobjekt för varje icke-numeriskt argument. Den metoden har dock ett stort antal uppenbara problem: det är alldeles för krångligt att utföra denna procedur för varje icke-numeriskt argument, och det skapar stor minnesfragmentering om man ska använda en hel sida för t.ex. en liten sträng på 20 byte.

Därför inför jag ett kernelobjekt, som jag kallar för *minneshål*. Detta kan tyckas vara ett fulhack, men med dagens PMMU:er ser jag ingen annan möjlighet. Ett minneshål är väldigt enkelt till sin funktion. Ett program skapar det genom ett systemanrop, CRHOLE, till vilket programmet skickar startadressen, längden av minnet det vill dela ut, samt en skrivskyddsflagga. Programmet skickar sedan minneshålet över porten det vill anropa, och tråden på andra sidan kan göra ett systemanrop, HOLECPY, för att kopiera innehållet på andra sidan hålet till en adress i den lokala kärnan eller vice

versa. Det senare förutsätter att hålet inte är skrivskyddat. Ännu ett systemanrop finns, som returnerar längden av det minne som delas ut av hålet: HOLELEN.

2.8 Schemaläggning

Schemaläggning (*scheduling*) är en av de aspekter av vanliga operativsystem idag som är minst flexibla. Även bland många mikrokernel-system tillhandahåller kernelen en hårdlödd schemaläggnings-algoritm, som program och användare kan tycka vad de vill om – de kan ändå inte göra något åt den.

Jag anser att schemaläggningen är en av de viktigaste aspekterna av ett operativsystem som bör vara så flexibel som möjligt. Ett batch-system vill förmodligen ha en helt annan schemaläggning än vad ett interaktivt system vill ha. Om ett system kör ett traditionellt grafiskt användargränssnitt med överlappande fönster är det inte omöjligt att man vill att det program som äger det aktiva fönstret ska ha högst CPU-prioritet. Man vill också förmodligen ha kortare time slices för att förbättra systemets interaktivitet. Ett beräkningssystem, å andra sidan, vill förmodligen ha långa time slices för att minska den prestandaförlust som korta time slices innebär. Ett time-sharing-system vill förmodligen ha en schemaläggare som ger alla användare lika CPU-tid, oavsett hur många trådar en viss användare har igång samtidigt. Möjligen vill man att varje inloggad användare dessutom får schemalägga sin egen CPU-tid som han/hon behagar.

Därför anser jag att det är väldigt viktigt att man ska kunna anpassa schemaläggningsalgoritmen. Kernelen har en inbyggd schemaläggningsalgoritm som är aktiverad när systemet bootar, vilken helt enkelt kör de trådar som är körbara i tur och ordning. Den lägger dock in en administrativ port i initieringsprogrammet. Denna port kan användas av initieringsprogrammet för att skicka en schemaläggnings-port till kernelen. Om initieringsprogrammet väljer att göra det, inaktiverar kernelen sin inbyggda algoritm. Istället anropar den den port som initieringsprogrammet skickade varje gång en time slice tar slut. Initieringsprogrammet kan sedan titta igenom listan över aktiva TID-nummer, välja den tråden som schemaläggningsalgoritmen tycker bör köras, och sedan anropa ett systemanrop, YIELDTO, till vilket det skickar ett TID-nummer och en tidsangivelse. Den tråd som anropar YIELDTO elimineras då av kernelen, och den tråd som har det angivna TID-numret får köra så länge som tidsangivelsen specificerar. Det finns naturligtvis inget som förhindrar att denna schemaläggningsalgoritm accepterar portar från användare som tillåter dem att schemalägga sina egna trådar själva.

2.9 Övriga systemanrop

Det krävs uppenbarligen ett sätt att skapa nya kärnor. För detta ändamål finns tre systemanrop[1]:

- CRCORE: Skapar en ny kärna utan några mappade områden och returnerar ett styrobjekt för kärnan.
- MAP2: Som MAP, men tar även ett OID-nummer som refererar till en kärnas styrobjekt, och mappar då in de begärda sidorna i den kärnan.
- CRGATE2: Som CRGATE, men tar även ett OID-nummer som refererar till en kärnas styrobjekt, och skapar då en port till den angivna adressen i den kärnan.

Ytterligare ett systemanrop tillhandahålls även för trådkontroll:

- RUN: Tar som argument en adress i den lokala kärnan, och skapar en ny tråd som börjar köra på den adressen. Returnerar ett OID-nummer som refererar till den nya tråden.

3 Grundläggande tjänster

I ett mikrokernel-baserat system räcker inte den grundläggande exekveringsmiljön till för att göra särskilt mycket. Därför bidrar operativsystemet även med att antal program som tillhandahåller vissa grundläggande tjänster som krävs för att kunna köra riktiga program.

Vissa av dessa tjänster, framför allt exec-servern är kraftigt inspirerade av GNU Hurd. GNU Hurd är FSF:s projekt för att skapa en fri UNIX-kernel, och bygger även den på en mikrokernel (närmare bestämt Mach). Jag vill därför kreditera Hurd-projektet för den inspiration det gett mig. Det finns inga särskilt auktoritativa källor för Hurds design, men projektets hemsida är `<http://www.gnu.org/software/hurd/>`.

I all sin lysande exemplariskhet är GNU Hurd dock långt ifrån perfekt. Dess största problem är just det faktum att den försöker emulera UNIX, med allt vad det innebär. De mest märkbara problemen är system-specifika användar-ID:n och att det bygger mycket på kernelens säkerhet, vilket gör det svårare att distribuera över flera datorer.

3.1 Drivrutiner

Drivrutiner brukar vara del av kernelen i de flesta operativsystem, även de flesta mikrokernel-baserade. Detta brukar kunna innebära ett antal problem:

[1] Proceduren med MAP2 och CRGATE2 känns ganska oelegant, men den gör sitt jobb. Bättre förslag mottages dock tacksamt

- Om en drivrutin är buggig och kraschar, brukar den kunna ha en tendens att dra med sig hela systemet i fallet, då den har fulla privilegier att skriva över kernelminne.
- Om man vill att systemet ska kunna ladda in nya drivrutiner under körning – vilket man allt som oftast vill – måste kernelen ha ett speciellt gränssnitt för att ladda in extra kod i moduler och manipulera dessa moduler, vilket leder till extra komplexitet i kernelen.

För att undvika dessa problem, anser jag att det bästa är att köra så många drivrutiner som möjligt som användarprogram. Efter observationen att de allra flesta drivrutiner bara kräver åtkomst till vissa speciella minnessidor och möjligen en avbrottshanterare, tror jag att det är möjligt att göra så med vissa gränssnitt. Om en drivrutin kraschar, kan det naturligtvis fortfarande orsaka problem, eftersom den hårdvaran den kommunicerar med kanske kraschar i sin tur och låser upp systembussen eller dylikt, men jag anser att denna arkitektur fortfarande kan komma att öka systemets stabilitet i allmänhet, och dessutom gör det lättare för utvecklare att experimentera med nya drivrutiner. Dessutom är det en säkerhetslättnad, eftersom drivrutiner inte måste få kernelprivilegier. Det innebär att vanliga användare kommer att kunna installera drivrutiner som inte innebär en risk att krascha hela hårdvaran, t.ex. drivrutiner för USB-enheter.

När kernelen bootar, ger den en administrativ port till initieringsprogrammet, med hjälp av vilken program kan göra anrop till kernelen för att få en mappare för sidor med angivna fysiska adresser. Likaså kan de, med hjälp av den porten registrera en port till kernelen som ska anropas när en avbrottsförfrågan kommer in från hårdvaran. För de få drivrutiner som faktiskt behöver mer tillgång än så, kan porten även utnyttjas för att flagga en kärna så att trådar i den körs i kernelläge. Om man låter initieringsprogrammet härska över denna port, och bara skicka nödvändiga mappare till de faktiska drivrutinerna kan man se till att drivrutinerna inte har mer hårdvaruåtkomst än de verkligen behöver.

I och med att systemet bootar hjälper initieringsprogrammet till att starta upp de drivrutiner som krävs för att kunna fortsätta boota, t.ex. diskdrivrutinen och konsolldrivrutinen.

3.2 Säkerhet

Säkerhet är en av de viktigaste grundstenarna i vilket operativsystem som helst – särskilt idag, med den otroliga explosion i cracker-samhället som skett det senaste decenniet och fortfarande fortsätter.

Det bör noteras att kernelen i sig inte tillhandahåller några som helst säkerhetsfunktioner, förutom åtkomstseperation i och med det faktum att den inte tillåter en kärna att komma åt andra kärnors objektreferenser. Säkerheten implementeras istället, som det mesta andra i systemet, på användarnivå.

Ett centralt begrepp är skillnaden mellan autentisering och auktorisering, en särkskiljning många andra system missar. *Autentisering* går endast ut på att avgöra *vem* det är man kommunicerar med. *Auktorisering*, å andra sidan, går ut på att avgöra *vad* den man kommunicerar med faktiskt får göra. Auktorisering kräver alltså autentisering, medan autentisering inte kräver auktorisering (även om det kan tyckas onödigt om en auktorisering inte följer).

3.2.1 Autentisering

Autentiseringsmodellen bygger på autentiseringsprotokollet Kerberos 5 [4]. Kerberos 5 definierar en form på användarnamnen, samt ett protokoll för att autentisera två användare mot varandra, med hjälp av en central autentiseringsserver.

Användarnamn i Kerberos 5 tar formen `namn/instans@OMRÅDE`. `OMRÅDE` (REALM på engelska) är det administrativa området som ansvarar för användaren. `namn` är namnet på användaren. `instans` är valfri och är (om den används) en vidare specificering av användaren. Den sistnämnda är väldigt ospecificerad i standarden vad den är tänkt att användas till, men i allmänhet går det ut på att skilja på olika användningar av användarnamnet. T.ex. kan jag skapa mig en användare som heter `fredrik/admin` för att göra det möjligt att separera vissa mer privilegierade funktioner att kräva att en mer skyddad inloggning används för att använda dem.

Den största fördelen med Kerberos 5 är att användarnamn är globala – med andra ord går de att använda över hela världen, förutsatt att båda parterna som vill använda dem är hopkopplade med varandra (t.ex. via Internet).

När en användare loggar in med Kerberos 5 får denne en krypterad ”biljett” som identifierar användaren i autentiseringsändamål. När två användare vill autentisera sig mot varandra kontaktar de en central autentiseringsserver – en KDC (Key Distribution Center) – som de båda litar på, vilken tar emot bådas biljetter och verifierar deras giltighet.

I systemet kommer alla komponenter – oavsett om de är användareprogram eller systemprogram – att ha en biljett för ett användarnamn. Eftersom systemprogrammen inte ”loggar in”, måste de uppenbarligen ha en annan mekanism för att erhålla biljetten – mer om detta i sektion 3.4.

Alla program som startas får en port till systemets autentiseringsserver (i princip en KDC, men med extrafunktioner för att t.ex. kunna autentisera över portar istället för över UDP, som Kerberos 5 vanligtvis gör). Systemets standardbibliotek tillhandahåller sedan en bekvämlighetsfunktion för att begära att en port autentiserar sig sig, och kommunicerar då med standardbiblioteket på andra sidan porten samt autentiseringsservern för att utföra en Kerberos-5-autentisering över porten.

3.2.2 Auktorisering

Det finns inte mycket allmänt att säga om auktorisering, eftersom auktorisering alltid är väldigt specifik för programmet som ska auktorisera. Tanken är dock att systemets standardbibliotek ska tillhandahålla en funktion för kontroll av en autentiserad användare mot en åtkomstlista. Åtkomstlistan är speciellt avsedd att lagras på disk i en fil. Att ha ett standardiserat filformat för åtkomstlistor är viktigt eftersom det innebär att mer eller mindre hela systemets auktorisering kan administreras med samma verktyg.

3.2.3 Övriga aspekter och implikationer

Det är min åsikt att denna kombination skapar ett säkerhetssystem som är mycket bättre än det i andra populära operativsystem idag. Ett av de absolut vanligaste säkerhetsproblemen i dagsläget är att en bugg i ett program som körs som administratörsanvändaren kan innebära att en som attackerar systemet kan få mer eller mindre fulla privilegier. Detta är särskilt sant för Unix, där många av systemprogrammen körs som root, och root har fullständig kontroll över hela datorn, men gäller även för många andra system, som Windows®. I det här systemet, å andra sidan, finns över huvud taget ingen administratörsanvändare. Dessutom gör det faktum att varje systemkomponent kör som olika användare system säkrare, eftersom ifall en komponent går att attackera, innebär det fortfarande väldigt begränsade privilegier – den som attackerar får alltid exakt de privilegier som det program som går att utnyttja på det sättet.

Vidare gör Kerberos 5 att administratörsfunktioner går att separera. Den användare som är administratör kan se till att alla adminstrativa funktioner bara är auktoriserade om han/hon är inloggad med `.../admin`-instansen. Även administratören kan alltså använda systemet som en vanlig användare utan att behöva vara rädd för sina extra privilegier.

3.3 Filsystem

Filsystemet är ett av de mest kontroversiella områdena inom operativsystemsdesign, och även det området där det är mest sannolikt att man kan få höra många väldigt olika åsikter om man går och förhör sig lite. Eftersom filsystem är ett så fantastiskt stort ämne i sig, tänker jag inte beröra det i djup i den här rapporten.

Eftersom ett av problemen jag vill lösa med det här systemet är flexibilitet, anser jag att enda sättet att designa en filsystemstjänst på är att specificera ett så generellt gränssnitt som möjligt, som är både simpelt och komplett i den meningen att det innehåller ett fåtal operationer, med vilka det går att göra så mycket som möjligt. Ett sådant gränssnitt bör också lämna så mycket frihet som möjligt till den bakomliggande implementationen. För att komma fram till vad för sorts gränssnitt man vill ha måste man

studera vad ett filsystem faktiskt är, och vilka funktioner man vill ha från det.

3.3.1 Fil

Den mest populära uppfattningen om en fil är att den är en uppsättning bytes i sekventiell ordningsföljd. Vilka funktioner man ska kunna anropa på en fil skiljer sig, men i regel går dessa funktioner att ordna upp i lager. Det första lagret är att kunna behandla filen som en byteström, d.v.s. läsa bytes från eller skriva bytes till strömmen sekventiellt. Vissa filer stannar på det lagret, t.ex. sådana som är ett gränssnitt mot en TCP-ström eller någon speciell hårdvaruenhet. Nästa lager är i regel en sökbar byteström, där man har lagt till en sökoperation, med vilken det går att flytta en pekare i filen fram eller tillbaka. Filer som stannar på det lagret kan t.ex. vara en fil som är ett gränssnitt mot en bandstation. Nästa lager är i regel att man kan ställa in storleken på filen – antingen minskar man filens storlek så att bytarna på slutet försvinner, eller så ökar man filens storlek så att man tillför bytar på slutet av filen. De filer som faktiskt lagras av filsystemet kommer att vara implementerade på det lagret. Det är ofta en önskedröm bland programmerare att tillföra ännu ett lager, vilket skulle medföra en funktion för att lägga till bytar var som helst i filen, så att de bytar som ligger efter flyttas fram för att göra plats. Det har dock visat sig så svårt och/eller ineffektivt att implementera det lagret i ett filsystem att inte ett enda operativsystem än så länge har implementerat ett sådant filsystem. Ett gränssnitt mot en fil bör alltså implementera ett antal av dessa funktionsanrop, beroende på vilket lager den är kompatibel upp till:

- READ: Läser ett antal byte från en ström. Tar som argument ett minneshål att lagra de lästa bytarna i, samt hur många bytar som ska läsas.
- WRITE: Skriver ett antal byte till en ström. Tar argument i analogi med READ.
- SEEK: Ställer in positionspekaren i en fil. Tar som argument den nya positionen i filen.
- TRUNCATE: Ställer in storleken på en fil. Tar som argument hur många byte filen ska innehålla.

Filer lagrade på disk kommer dessutom att ha följande viktiga funktionsanrop:

- CRMAP: Skapar och returnerar en mappare som, vid mappning, skapar minnessidor som motsvarar filens innehåll, i enlighet med MMAP i UNIX.

3.3.2 Struktur

Ett filsystem är en uppsättning persistent lagrade (persistent i betydelsen att lagringsformen klarar av en omstart) filer samt en persistent lagrad struktur som beskriver en relation mellan filerna. Det vanliga sättet att konstruera strukturen i dagsläget är att flagga vissa filer som *kataloger*, vilka innehåller länkar till andra filer (eller kataloger) samt namn på dessa länkar. Detta tycker jag är fel, eftersom det sammanlänkar filer och struktur i en enda enhet, medan de egentligen inte har något med varandra att göra – jag ser hellre att man separerar strukturen från filerna. Om man separerar de tu, är strukturen mellan filerna ej längre låst till filernas interna struktur, och därför mer flexibel. Det ger också mer frihet till den faktiska implementationen av filsystemet: Filsystemsprogrammet kan välja att flagga filer och använda till att lagra strukturen om det så vill, men det kan lika gärna låta bli, och istället göra på något helt annat sätt.

Strukturen kräver dock ett väl definierat gränssnitt. Den struktur jag har valt att definiera är en riktat graf (ej ett riktat träd, som de flesta operativsystem väljer att implementera). Strukturen består av ett (vanligtvis stort) antal noder. Var och en av dessa noder har ett (vanligtvis litet) antal namngivna namnrymder. Var och en av dessa namnrymder innehåller en lista innehållande länkar till andra noder, samt namn på dessa länkar. Namnrymdernas syfte är att ge möjlighet till flera substrukturer. De namnrymder som kommer vara standardiserade är ”`struct`” samt ”`attr`”. Namnrymden ”`struct`” är egentligen den som faktiskt utgör filsystemets struktur. ”`attr`” är reserverad för möjligheten att kunna lagra namngivna attribut för en fil, t.ex. en beskrivning av dess innehåll, åtkomstlistor o.s.v. Jag väntar mig egentligen inte att det kommer krävas fler namnrymder, men jag bibehåller gärna möjligheten, för flexibilitetens skull. Varje nod har dessutom den valfria möjligheten att länka till en fil. Om en nod har en fil associerad med sig, är ett anrop, OPEN, tillåtet, vilket kontrollerar anroparen mot nodens åtkomstlistor, och returnerar en port motsvarande filen om åtkomstlistorna tillåter det.

Jag vill även smyga in ännu en Hurd-inspirerad funktion i filsystemet, nämligen översättare (*translators* på engelska). Tanken är att OPEN egentligen går att utföra om en av två möjliga villkor är uppfyllda: antingen om en fil är associerad med noden, eller om ett attribut vid namn `trans` är satt på noden. Om det senare är sant, länkar det attributet till en annan fil, vilken startas som ett program och returnerar en port som uppfyller gränssnittskraven för en fil. På så sätt ökas flexibiliteten genom att man kan ha ”virtuella” filer.

Gränssnittet mot en strukturnod måste alltså ha åtminstone följande anrop:

- FOLLOW: Letar upp länken med angivet namn i angiven namnrymd, och returnerar en port som motsvarar den strukturnod som återfinns i

andra änden av länken.

- LIST: Enumererar länkarna i den angivna namnrymdem.
- OPEN: Beskrivet ovan.
- LINK: Tar en port till en annan strukturnod och skapar en länk i en angiven namnrymd med ett angivet namn som syfter till den noden som angavs.
- UNLINK: Tar bort en länk som specificeras genom namn och namnrymd.
- ASSOC: Associerar noden med en fil, vilken specificeras genom att en port till filen skickas med, eller ett NULL-argument för att avassociera den fil som för tillfället är associerad med noden.

Gränssnittet har också följande anrop, som egentligen syftar på filsystemet i stort snarare än en specifik nod:

- CRFILE: Allokerar en ny fil som inte är associerad med någon nod och returnerar en port som motsvarar den filen.
- CRNODE: Allokerar en ny nod och returnerar en port som motsvarar den.

3.3.3 Sökvägar

Filsystemsservern i sig själv har ingen uppfattning om sökvägar. Sökvägsfunktioner tillhandahålls istället av standardbiblioteket som en bekvämlighetsfunktion. Min tanke är att sökvägsformatet ska vara UNIX-liknande, med snedstreck som elementseparator, och inledande snedstreck för att indikera en absolut sökväg. Absolut sökväg kräver dock vidare klarifiering, i och med att man måste definiera en strukturnod som den första noden när en absolut sökväg tolkas. Denna nod får ett program i och med att det startas, av det programmet som startade det. Denna nod kommer från början från initieringsprogrammet, som får den av filsystemsservern i och med att systemet bootas. Denna nod kallas *CAN*, för Common Access Node. Standardbiblioteket håller själv reda på den aktuella noden för relativa sökvägar.

Jag vill även ta detta ett steg längre och definiera det som så att ett användareprogram även får en port till den nod som motsvarar en ”hemkatalog”. Sökvägsfunktionerna i standardbiblioteket tillåter en sökväg att vara relativ till denna nod, återigen med UNIX-syntax: en sökväg som inleds med tilde och snedstreck är relativ till hemnoden. Detta har framförallt fördelen att hemnoden kan ligga på ett helt annat filsystem, och behöver inte nödvändigtvis vara möjlig att nå via en absolut sökväg.

Var god notera hur ett program som startar ett nytt program inte nödvändigtvis behöver ge det nya programmet dessa portar. Detta är ett sätt

att lösa vanliga säkerhetsproblem – om man vill köra ett program man inte litar på, t.ex. ett som kommer direkt från någon konstig sida på Internet, kan man låta bli att ge det dessa portar, och det kommer då inte ha tillgång till filsystemet över huvud taget.

Jag planerar även att standardbiblioteket bör tillhandahålla funktioner för tolka och ”öppna” URL:er.

3.3.4 Administration

En absolut nödvändig funktion för ett filsystem är (tyvärr) att låta en administratör ha fullständig kontroll över det, oavsett vad åtkomstlistorna säger. För detta syfte har filsystemsservern en speciell åtkomstlista som avgör just vilka som har det privilegiet. Administratören kan då lägga in sin egen användare däri. Det är naturligtvis att rekommendera att administratören skapar sig en särskild Kerberos-instans för denna uppgift, för att undvika misstag och säkerhetshål.

3.4 Exec-servern

När ett program startas får det en port till en exec-server. Exec-servern kan sedan anropas just för att starta ett nytt program. För att göra detta anropar ett program funktionen EXEC genom exec-serverns port, till vilken den skickar en port rill en strukturnod i filsystemet som motsvarar den fil som innehåller programmet som ska startas. Exec-servern skapar då en nya kärna, mappar in programmet i den och utför övriga nödvändiga funktioner för att konstruera och köra programmet. Till sist skapar exec-servern en tråd i den nya kärnan för att låta programmet köra.

Mycket av detta skulle förvisso kunna utföras av standardbiblioteket, men det finns en viktig orsak till varför man behöver exec-servern: den har möjligheten att starta ett program som en annan användare (i stil med SUID-program i UNIX). Naturligtvis har inte exec-servern några särskilda privilegier för att bara kunna anta vilken användare som helst – när ett sådant program installeras skapar användaren som installerar det en speciell länk i `attr`-namnrymden, som går till en annan strukturnod som innehåller relevanta länkar för att exekvera ett sådant program. Endast den användaren som exec-servern kör som (och användaren som installerade programmet) har läsrättigheter till denna nod, vilken innehåller en fil som anger vilken Kerberos-användare det nya programmet ska loggas in som (förslagsvis en annan instans än användarens vanliga inloggningsinstans), Kerberos-nyckeln som krävs för att logga in som den användaren, och slutligen ett antal valfria länkar som den som installerade programmet tyckte skulle vara bra att ha. CAN-noden är ett bra förslag att ha bland de länkarna, men de är egentligen helt godtyckliga.

4 Distribution

Även om jag anser att det system jag just beskrivit har många fördelar över nuvarande populära system även för enanvändarsystem, kommer dess verkliga fördelar fram när man har flera datorer hopkopplade över nätverk eller, än bättre, över Internet.

Distribuerade system är förvisso inget nytt, och många av dem har vissa fördelar över detta. Alla system jag har studerat än så länge har dock en gemensam brist: om två system, distribuerade eller inte, ska kommunicera med varandra sker det inte längre med systemets interna distribueringsprotokoll – det blir som två vanliga nätverksoperativsystem som kommunicerar med varandra.

Förvisso finns det system som delvis tillåter användare att bryta ner datorgränserna. Det vanligaste exemplet är förmodligen Unix/Linux med NFS och NIS/LDAP, eller Windows-system som är del av en, på ett sådant sätt administrerad, Windows-domän. Detta tillåter användare att logga in på vilken dator som helst inom systemet. Det måste dock noteras att även i detta fall kvarstår gränserna mellan två olika system, eftersom, framförallt i fallet Unix/Linux (gäller även för Windows, men det syns inte lika väl där, eftersom de detaljerna sällan når användarens ögon), de användar-ID:n som används internt av systemet endast är giltiga inom det lokala administrativa området. Därför går det inte, ens med de metoderna, att logga in utanför systemet.

Vidare finns filsystemet AFS (samt andra som tagit efter idén – Coda, NFSv4, m.fl.), vilket definierar ett globalt filsystem, vanligtvis i och med alla sökvägar inom det globala filsystemet inleds med namnet på det administrativa området. Även med detta kvarstår dock många gränser mellan olika system, i och med det att, även om man kan komma åt sina filer transparent på andra system, kan man fortfarande inte logga in på ett annat system med sitt vanliga användar-ID, då detta fortfarande är system-specifikt. Detta innebär att det fortfarande finns en mängd saker man inte kan göra mellan två olika system även med t.ex. AFS, såsom att köra ett program på en annan dator transparent eller, i synnerhet, att faktiskt *logga in* på en dator utanför systemet.

4.1 Proxying

Anledningen till att det här systemet lämpar sig bra för distribuering är att alla kernelobjekt som används för kommunikation mellan kärnor har valts noggrant för att de ska vara nätverkstransparenta. Till exempel: Om ett program har en port till ett annat program, gör det totalt detsamma om porten går direkt till det andra programmet, eller om det går till ett helt annat program – en *proxy* (jag saknar en bra svensk översättning) – som inte bryr sig om vad argumenten som skickades över porten faktiskt innebär,

utan bara enkodar dem med något lämpligt protokoll, och skickar dem över nätverket till en annan dator, på vilken andra sidan av proxyn körs, som tar emot de enkodade argumenten, dekodar dem, och utför ett till portanrop till det programmet som faktiskt ska utföra den efterfrågade operationen. Detta gäller naturligtvis alla program som använder portkommunikation, inklusive, men icke begränsat till, filsystemsservern, exec-servern eller en drivrutin. Även mappare och minneshål är nätverkstransparenta. Det svåraste att se är antagligen hur en mappare kan vara nätverkstransparent, men jag har kommit på en mekanism för att göra det på användarnivå, vilken jag dock inte tänker gå in på i detalj här. Man bör dock notera att man bör vara försiktig med att använda en mappare över nätverket, eftersom minne brukar användas på ett sätt som förutsätter mycket lägre latens än vad man i regel har över ett nätverk – inte minst Internet. Så länge inget program skriver till det mappade minnet bör det dock inte vara några större problem.

4.2 Datorgränser

Det ska erkännas att det här operativsystemet inte är lika transparent som andra distribuerade system, som t.ex. ett Beowulf-kluster, i och med att det är svårt (om än inte omöjligt) att migrera en kärna från en kernel till en annan. Detta är dock halvt avsiktligt – jag anser att de flesta distribuerade system faktiskt har för svaga gränser mellan datorer. Även om detta har fördelar om man vill bygga en superdator av 2.000 vanliga PC-datorer, har det ännu större nackdelar när man försöker integrera alla datorer i ett helt nätverk i ett system. Detta är på grund av att alla datorer i ett nätverk sällan är så likställda som 2.000 PC-datorer i ett kluster är. Man har oftast ett antal servrar, med olika hårdvara för olika funktioner (t.ex. en server med en hel massa lagringsutrymme för att serva filsystem, en server med en hel massa processorer för beräkningsintensiva program, o.s.v.), ett antal arbetsstationer, och eventuellt bärbara datorer, mobiltelefoner, skrivarservrar, m.m.

För det första har alla dessa datorer helt olika prestandaprofiler – man vill t.ex. gärna köra beräkningsintensiva program på en server med många och/eller snabba processorer på, medan man gärna kör en editor eller andra interaktiva program på sin arbetsstation, så att man slipper nätverkslatens i användargränssnittet. Än viktigare är dock säkerhetsimplikationerna – om man sitter vid en arbetsstation vill man under inga omständigheter att ett av ens program migreras över till någon annans arbetsstation. Eftersom varje användare av en arbetsstation har direkt tillgång till hårdvaran och därför kan göra en massa bus, vill man hemskt gärna att ens program antingen körs på den arbetsstationen man sitter vid, eller på en server som man vet är säker i en datorhall någonstans nere i källaren. Detta gäller framförallt SUID-program – systemadministratören bör tilldela exec-servrar på olika datorer olika Kerberos-instanser, t.ex. `exec/trusted` och `exec/workstation`,

för att se till att SUID-program kan begränsas till att endast köras av exec-servrar på tillförlitliga datorer (genom att man begränsar läsrättigheterna till Kerberos-nyckeln till t.ex. `exec/trusted`).

Därför har det här operativsystemet en klar gräns mellan fysiska datorer, även om objekt kan delas ut transparent till andra datorer. Till exempel kan exec-servrar inom ett nätverk ställas in på att samarbeta för att köra ett program på den mest lämpliga datorn (beräkningsintensiva program på en dator med kraftfulla processorer, interaktiva program på användarens arbetsstation o.s.v., eller helt enkelt på den datorn som har minst att göra, eller någon av ett otal andra allokeringsstrategier).

4.3 Global inloggning och administrativa gränser

Den riktigt intressanta implikationen av systemets arkitektur kommer utav att Kerberos-användarnamn är helt globala[2]. På grund av detta finns inga förhinder att en dator som körs på ett Internet-café i Hong Kong autentiserar en användare på KTH:s system och loggar in denne – precis som om han/hon loggade in lokalt på KTH – utan att någondera av de inblandade administratörerna behöver göra något särskilt för att det ska funka. Det är just denna form av *global inloggning* som jag anser gör systemet så speciellt att det bör kallas för ett femte generationens operativsystem; det är inte ett distribuerat system – det är ett globalt system.

Naturligtvis vill man inte alltid att global inloggning ska vara möjligt – om KTH köper en ny dunderdator för att analysera ett par strängteoretiska vågfunktioner, kanske de inte vill att den lokala Counterstrike-klanen loggar in på den och kör 50 dedikerade servrar för en turnering de ska ha. Eftersom både CPU-tid och minne är begränsade resurser, bör man kräva auktorisering även för dem, d.v.s. olika inloggningsprogram bör använda åtkomstlistor för att kontrollera huruvida en viss användare får logga in på ett visst system. Var god notera att dessa administrativa gränser är valfria och godtyckliga, och det vore t.ex. vist för Internet-cafén att tillåta användare från andra områden att logga in lokalt på deras arbetsstationer. Likaså kan ett system begränsas totalt till bara en dator om man så önskar.

[2]Förvisso krävs det särskilda åtgärder vid autentisering mellan olika områden (*cross-realm authentication*), men jag har förhört mig med Kerberos-experter som försäkrar mig om att det inte finns något problem med att uppfinna en mekanism för att automatiskt sätta upp sådan autentisering när det behövs.

5 Argumentation

5.1 Potentiella anmärkningar

5.1.1 Kvalitativa anmärkningar

De främsta kvalitativa nackdelarna med det system jag beskrivit ligger i det faktum att det fortfarande finns märkbara gränser mellan olika fysiska datorer – i synnerhet kan inte kärnor migreras mellan olika datorer. Att kunna migrera kärnor vore väldigt bra ur framför allt tillgänglighetssynpunkt – om man behöver stänga av en server för underhåll, skulle man kunna migrera över alla kärnor till en annan server, och ingen klient skulle märka någon skillnad. Det bör dock noteras att de kvarvarande datorgränserna är, till största delen, avsedda, framför allt av administrativa skäl, t.ex. för att kunna förhindra global inloggning där den är oönskvärd, samt att det ofta är användbart att kunna utnyttja den struktur som datorgränserna skapar. En klustrad mikrokernel till detta system (se sektion 5.2.2) skulle också kunna motverka en stor del av, om inte alla, dessa problem. I ljuset av dessa två punkter anser jag att det resultatet helt enkelt är värt dessa datorgränser.

De näst främsta kvalitativa nackdelarna finns framför allt i filsystemsstrukturen. Det finns många alternativ till en struktur som är byggd utifrån en riktad graf med namngivna länkar, och de är absolut inte utan fördelar. Som ett specialfall bör även nämnas att det faktum att den filsystemsstruktur jag beskrivit tillåter cykler innebär att man behöver en skräpinsamlare för att hitta avlänkade sådana cykler, och en skräpinsamlare för filsystemet kan innebär stora prestandaförluster, eftersom den kan komma att ta upp väldigt mycket bandbredd och söktid från disken. Allt i allt måste jag erkänna att filsystemsstrukturen är den punkt jag i dagsläget är minst säker på, och jag är öppen för förändringar i den. I synnerhet ämnar jag läsa det som Hans Reiser har att säga på `<http://www.namesys.com/whitepaper.html>`. Jag misstänker att jag kommer att ändra strukturen efter att ha läst det, då det lilla jag ännu läst låter väldigt lovande. Jag vill dock framhålla att filsystemet fortfarande ska skilja mellan struktur och faktiska filer, och bli kvar vid den modellen att man länkar filer till strukturnoder, av de anledningar jag redan beskrivit i sektion 3.3. Det bör noteras att filsystemsstrukturen inte ännu är en kritisk del av systemet, och fortfarande kan ändras utan att det påverkar andra delar.

5.1.2 Kvantitativa anmärkningar

Inte mycket kan ännu säkert sägas om systemets kvantitativa aspekter, då det ännu inte finns någon implementering av det. Man kan dock identifiera vissa potentiella kvantitativa problem.

Det främsta kvantitativa problemet som kan tänkas uppkomma är prestandaproblem p.g.a. nätverkslatens. Eftersom varje RPC-anrop som utförs

över en nätverksproxy först måste skicka sina argument över nätverket, och sedan vänta på att resultaten återvänder, är detta mycket susceptibelt för problem som orsakas av nätverkslatens – i synnerhet i samband med sådana operationer som kan tänkas behöva utföras många gånger, sekventiellt, och över höglatensnätverk som Internet, framför allt filsystemsåtkomst. Att hämta innehållet i 1000 filer över en Internet-länk som har en latens på 50 ms skulle ta åtminstone $1\frac{1}{2}$ minut. Detta är naturligtvis inte godtagbart på någon nivå. Dock måste noteras att de RPC-anrop, som portkommunikationen tillhandahåller, är den *generella* kommunikationskanalen i systemet – den finns alltid tillgänglig, även om den kanske inte har världens bästa prestanda. Det finns dock ingenting som förhindrar att man skriver specialiserade proxyprogram för särskilda funktioner, såsom filsystem. Man kan t.ex. tänka sig att man skriver en filsystemsproxy som utför enkelt cachelagring på klientsidan, för att minska nätverkskommunikationen. Man kan även tänka sig en mer avancerad proxy som även utför filsystemssynkronisering (alltså helt och hållet kopierar väl utvalda filer till den lokala disken), t.ex. för användning med bärbara datorer, mobiltelefoner, m.m. som inte alltid har tillgång till något nätverk.

Vidare kan man anmärka om schemaläggningsalgoritmen, att den kan komma att ta upp för mycket processortid bara för att schemalägga, eftersom den potentiellt kan komma att gå igenom många portar bara för att komma fram till vilken tråd som slutligen ska köras. Detta kan vara ett problem. Även om de flesta moderna datorer klarar av många miljoner trådbyten per sekund, är det inte säkert att det gäller för alla arkitekturer. Det bör dock noteras att den anpassningsbara schemaläggningen inte är nödvändig, bara fördelaktig. På arkitekturer där det kan komma att bli ett problem, kan man helt enkelt låta kernelen fortsätta gå med sin inbyggda schemaläggningsalgoritm (se sektion 2.8). Även om det naturligtvis inte är att föredra, är det i alla fall ett sätt att råda bot på situationen. Dessutom kan man tänka sig att man kompromissar på ett sådant sätt att de underliggande kärnor som skulle få sköta om mer finkornig schemaläggning, bara uppdaterar prioriteten på sina trådor då och då, och skickar information till initieringsprogrammet om hur de vill ha sina trådar schemalagda. På det sättet kan hela schemaläggningen skötas direkt från initieringsprogrammet på ett sätt som är mycket mer flexibelt än många av dagens system, om än inte lika flexibelt som en fullständig implementation av den schemaläggning som beskrevs i sektion 2.8.

5.2 Jämförelse med andra system

Det finns naturligtvis andra system som är i experimentfasen för tillfället. Jag ska försöka presentera ett antal av dem, som jag lyckats skaffa mig information om, samt deras för- och nackdelar. Det finns även de som är svårt att få information om, tyvärr. Det främsta av dessa är ett operativsystem

som har fått namnet Unununium (av det temporära namnet för det 111:e grundämnet). Dess designmål ser lovande ut – även om jag inte håller med om vad de säger om ortogonal persistens[3] – men det är mycket svårt att hitta någon form av djupare information om det förutom genom att titta på källkoden. För de som är intresserade, finns projektets hemsida att finna genom URL:en `<http://unununium.org/>`.

När det gäller de system jag kan jämföra med, måste det noteras att jag inte kan göra några kvantitativa jämförelser (vad gäller prestanda, minnesåtgång o.dy.), eftersom dessa faktorer är beroende av implementationen av ett system. Eftersom detta system (ännu) saknar en implementation, blir dessa jämförelser omöjliga. Jag måste därför begränsa mig till rent kvalitativa jämförelser.

5.2.1 Nätverksoperativsystem

Om man gör en kvalitativ jämförelse mellan de vanliga nätverksoperativsystemen av idag (alltså t.ex. Microsoft Windows, Linux, *BSD, Mac OS X, andra UNIX-varianter, o.s.v.) och detta operativsystem, vill jag mena på att det går att strikt bevisa att detta operativsystem bara är bättre.

Det må låta överdrivet, men jag anser att det är på detta vis, eftersom all funktionalitet som finns i dessa operativsystem (i *systemet* alltså – applikationsprogram o.dy. kan jag naturligtvis inte diskutera i denna rapport) finns i det system jag beskrivit. Utöver detta finns en hel del mer funktionalitet, som t.ex. globala symboliska användarnamn, anpassningsbar schemaläggning, mappare, ett flexibelt filsystem, mikrokernel, drivrutiner som körs i användarläge, m.m. Ett strikt bevis kräver att man går igenom varje del funktionalitet i dessa system och ser till att den finns tillgänglig, men ett sådant bevis skulle bli på tok för långt för att kunna inkluderas i denna rapport.

Av den anledningen menar jag på att det system jag häri beskriver endast har fördelar jämfört med dagens vanliga nätverksoperativsystem, kvalitativt sett. Det innebär även att man med relativ enkelhet kan implementera dessa systems API:er som ett bibliotek, för att tillhandahålla binär kompatibilitet med dem. Enklast är förmodligen UNIX-liknande system, då glibc kan portas med relativ enkelhet.

[3] Unununium-folket ser RAM-minnet som en cache av hårddisken, och hävdar därför att om man bara kan synkronisera hårddisken mot RAM-minnet med hjälp av samma tekniker som används av journaliserande filsystem, kommer man kunna stänga av datorn när som helst för att sedan sätta på den igen och fortsätta där man var. Jag håller inte med om detta, eftersom RAM-minnet har en helt annan semantisk funktion än vad hårddisken har – RAM-minnet är del av datorns kärnarkitektur, tillsammans med CPU:n, systembussen och ingenting mer. Hårddisken är en fullständigt valfri periferienhet. Vill man åstadkomma persistens över omstarter, skall detta lösas på hårdvarunivå, t.ex. genom MRAM.

5.2.2 Klustersystem

Den klassiska idén om ett distribuerat system är det jag kallar för ett klustersystem. Själva tanken med ett klustersystem är att man sätter ihop ett antal datorer och får dem att se ut som ett enda klassiskt time-sharing-system. Exempel på klustersystem är Amoeba, klustrat Linux (med MOSIX, Beowulf, m.fl.), klustrat Mac OS X (med Xgrid) och klustrat Windows (ska vara inbyggt på något sätt i de dyrare varianterna). För information om Amoeba, se [5].

Även om klustring är en god tanke i sig själv, är den otillräcklig. Själva den explicita idén är att få ett time-sharing-system med hög prestanda genom att få flera datorer att se ut som en. Däri ligger att man kör ett normalt time-sharing-system på denna virtuella dator, vilket genast utesluter den sortens global inloggning som beskrevs i sektion 4.3, och har i övrigt mer eller mindre alla de fördelar och nackdelar som ett sådant system har. I synnerhet innebär detta att de klustrade varianter av existerande operativsystem (Linux, Mac OS X, Windows, m.fl.) har samma flexibilitetsproblem som de oklustrade versionerna.

Som jag ser det är klustring ett komplement till det operativsystem jag beskrivit, snarare än en designparadigm i sig själv. Det vore väldigt trevligt att utveckla en specialversion av mikrokernelen till detta operativsystem som kan få flera fysiska datorer att köra som om de vore en dator som körde en mikrokernel, i och med att man skulle få sådan funktionalitet som lokalitetstransparens och migreringstransparens (se [1] för en förklaring av de termerna) av kernelobjekt. Den virtuella dator som denna klusterkernel skulle skapa vore dock en nod i det större globala system som möjliggörs av detta operativsystem, snarare än ett eget isolerat system, såsom dagens klustersystem skapar. Denna virtuella dator skulle sedan kunna vara fantastiskt användbar som t.ex. beräkningsserver.

5.2.3 JX

Det finns ett par projekt för att bygga ett operativsystem helt i Java. Det ena – JOS – verkar tyvärr ha dött för ett tag sedan. Det andra – JX – å andra sidan, tycks vara högst levande. JX är ett tämligen trevligt operativsystem – det kör på en mikrokernel, det har drivrutiner i användarläge (t.o.m. skrivna i Java) och det har en flexibel schemaläggare.

Problemet, som jag ser det, är just att det är skrivet i Java. Inte för att jag har något emot Java i sig självt (vilket jag i och för sig har, men det hör inte hit), men det är en onödig begränsning att *behöva* skriva sina program i Java. I det operativsystem jag beskrivit finns det ingenting som förhindrar en kärna från att köra ett program skrivet i Java, eller, ännu bättre, motsvara ett Javaobjekt som man kan köra Javametodanrop mot med hjälp av en port. En kärna som agerar på det sättet kan behandla andra kärnor i systemet som

andra Javaobjekt, och representera dem sådant för den Javakod som körs i den. Å andra sidan kan man *även* skriva sina program i C, Assembler eller vadhelst man vill och finner bäst för det jobbet man vill utföra.

Ett Javasystem ger dock en fördel, vilken ej bör underskattas, nämligen det att ett program skrivet till det kan köras på vilken hårdvara som helst, vare sig det är en Intel-arkitektur, Sun, PowerPC eller MIPS, så länge de grundläggande delarna av operativsystemet finns porterade dit. Det är en kvalitét som har tillräckligt många och stora fördelar för att vara värt att lösa för andra operativsystem likaså – särskilt för ett operativsystem, som detta som är tänkt att vara globalt, och därför även kommer spänna sig över många hårdvaruplattformar. En preliminär tanke som jag har för att komma till en lösning, är att man, om man önskar distribuera ett plattforms-oberoende program, skulle kunna distribuera en fil som innehåller den information som kompilatorn genererar precis innan den anropar kodgeneratorn för den specifika plattform man kompilerar för. Sedan, när programmet ska köras, kan exec-servern anropa kodgeneratorn för den aktuella plattformen för att generera ett körbart program. Detta är ju ungefär vad som sker med Java som det ser ut just nu i alla fall, så det borde fungera på ett rent teoretiskt plan. För att minska tiden det tar att starta ett program på detta sätt kan man tänka sig att man kan kompilera en funktion i taget, alltefter som de behövs.

Återigen bör noteras att distribution av program i denna form är *valfri* – man kan lika gärna distribuera det slutgiltigt kompilerade programmet istället. Det tycks mig dumt att aktivt förbjuda program från att vara skrivna direkt för en specifik processor. Det är av denna anledning jag fortfarande anser att JX gör systemet för begränsat, utan att tillföra några uppenbara fördelar – åtminstone inga som jag kan se.

Mer information om JX finns på dess hemsida, vilken finns att finna genom URL:en `<http://jxos.org/>`. Där finns även referenser till artiklar publicerade om JX.

Referenser

[1] *Modern Operating Systems*
Andrew S. Tanenbaum
Prentice Hall International 1992

[2] *A Comparison of Two Distributed Systems: Amoeba and Sprite*
Fred Douglis, M. Frans Kaashoek, et al.
Computing Systems, vol. 4, num. 4, sid. 353–384, 1991

[3] *The Mythical Man-Month: Essays on Software Engineering*
Frederick P. Brooks, Jr.
Addison Wesley International 1995 (Anniversary Edition)

[4] *IETF RFC 1510 – The Kerberos Network Authentication Service (V5)*
J. Kohl, C. Neuman
<ftp://ftp.rfc-editor.org/in-notes/rfc1510.txt>

[5] *The Amoeba Distributed Operating System*
Andrew S. Tanenbaum, Gregory J. Sharp
Vrije Universiteit 1992
<ftp://ftp.cs.vu.nl/pub/amoeba/Intro.ps.Z>