OS-X, ett distribuerat realtidsoperativsystem

(C) Francis Görmarker 1994

Detta dokument beskriver operativsystemet OS-X, ett kompakt, distribuerat realtidsoperativsystem för mikrodatorer.

1.0 Inledning 4
2.0 Processhantering 5
2.0.1 Inledning 5
2.0.2 Schemaläggning mm 5
2.0.3 InterProcess Communication 5
2.0.4 Start av process 5
2.0.5 Terminering av process 6
2.0.6 Remote Procedure Call 6
2.0.7 Standard RPC 7
3.0 Filsystem 8
3.0.1 Inledning 8
3.0.2 Universal File Identifiers 8
3.0.3 Directory server 9
3.0.4 File server 9
3.0.5 Systemstruktur 10
4.0 Memory server 11
4.1 Funktion 11
4.2 RPC-gränssnitt 11
5.0 Process server 13
5.1 Funktion 13
5.2 RPC-gränssnitt 13
6.0 Directory server 15
6.1 Funktion 15
6.2 RPC-gränssnitt 15
7.0 File server 17
7.1 Funktion 17
7.2 RPC-gränssnitt 17
8.0 Exec server 19
8.1 Funktion 19
8.2 RPC-gränssnitt 19
9.0 Name server 21
9.1 Funktion 21
9.2 RPC-gränssnitt 21
10.0 Network server 23
10.1 Funktion 23
10.2 RPC-gränssnitt 23
10.3 Länkprotokoll 24
10.4 Tillståndsbeskrivning 24

11.0 Drivrutiner 26
11.1 Inledning 26
11.2 Gränssnitt mot hårdvaran 26
11.3 Gränssnitt mot drivrutiner 27
12.0 Systemanrop 28
12.0.1 Minneshantering 28
12.0.2 Processhantering 28
12.0.3 IPC 30
12.0.4 Filhantering 31
12.0.5 Diverse 32
12.1 Inhoppsadresser 33

1.0 Inledning

OS-X är ett operativsystem för små mikrodatorsystem och finns för närvarande implementerat på några olika Z80-baserade system. Systemet har vissa influenser av UNIX men är ej avsett att på något sätt vara UNIX-kompatibelt. Några av de lånade koncepten är:

- Hierarktiskt, dynamiskt monterbart filsystem baserat på i-noder och mycket enkel katalogstruktur.
- Processhierarki med fork/join, ärvning av öppna filer mm.
- Hantering av IO-enheter via “special files” och IPC med hjälp av “sockets”.

De lånade koncepten är i många fall nedskalade för att bättre passa in i ett i huvudsak enanvändar- mikrodatorsystem. Processmodellen skiljer sig kraftigt ifrån UNIX-modellen, i OS-X är alla processer likvärdiga dvs det finns ingen “kernel mode”. Både tung- och lättviktprocesser kan skapas, tungviktsprocesser kan ha separata fysiska adressrymder och kan endast skapas med systemanropet “exec”, lättviktsprocesser skapas med hjälp av systemanropet “fork” och delar alltid adressrymd med sin förälder. IPC (Inter-Process Communication) sker via dynamiskt allokerade “sockets” och är baserad på meddelandesändning från punkt till punkt. Meddelandestorleken är begränsad till 256 bytes och kan utföras blockerande eller icke-blockerande vid sändning och mottagning.

Systemkärnan implementeras av ett antal procedurer och processer. Systemanrop utförs genom proceduranrop till ett antal lågnivårutiner i delat minne eller som RPC mot systemprocessen som hela tiden lyssnar på en välkänd “socket”. OS-X bygger på principen med en minimal kärna och ett antal server-processer som tillhandahåller filsystem, programexekvering mm. Då OS-X använder ett generellt RPC-baserat gränssnitt för filhantering, programexekvering mm så är det mycket enkelt att skriva distribuerade applikationer och att köra applikationer i distribuerad miljö.

2.0 Processhantering

2.0.1 Inledning

Processer av två typer: tungviktsprocesser med egen adressrymd och lättviktsprocesser som delar adressrymd med minst en tungviktsprocess. Lättviktsprocesser har alltid en tungviktsprocess som förälder och kan ej existera utan sin förälder pga att föräldern äger den delade adressrymden. Vid start av en lättviktsprocess så ärvs alla öppna filer. När en tungviktsprocess terminerar så termineras också alla dess lättvikts- subprocesser och den delade adressrymden avallokeras. Om en lättviktsprocess startar en ny lättviktsprocess så får den närmaste förfader av tungviktstyp som förälder. På detta sätt så har alltid lättviktsprocesser en tungviktsprocess som förälder.

2.0.2 Schemaläggning mm

Alla processer schemaläggs beroende på prioritet. Fyra prioritetsnivåer finns, 0-3 där noll är den lägsta nivån. Processer med samma prioritet shemaläggs med tidsdelning där varje process tillåts exekvera maximalt 8 ms. Processer med hög prioritet exekveras alltid före processer med lägre prioritet. Exempel: en process på nivå tre exekveras tills den terminerar eller blockeras innan någon process på lägre nivå tillåts exekvera. Lågprioriterad hold(), dvs återstart av en process efter hold fås genom att länka in processen sist i activeQ för aktuell prioritet. Timeout möjlig vid blockerande ipc_rec(). Varje process har en timer som räknas ner var 32:a tick (dvs var 256:e ms). När timern når noll lyfts processen ur aktuell kö och aktiveras. För aktiva processer sätts timern till noll vilket innebär att timeout ej används.

kill() är ett kernel RPC som används för att omedelbart terminera en process, eventuella lättviktiga subprocesser termineras också. För att hantera terminering av förgrundsprocesser från en terminal används en semafor och en högprioriterad systemprocess (breakd). När signal utförs på semaforen så aktiveras breakd-processen vilken letar upp och terminerar alla processer som har flaggan bflag = true. Flaggan bflag finns i processtabellen för alla processer och den sätts normalt = true då en ny process startas. Anropet set_break() används för att ställa om bflag. Processen breakd terminerar först alla berörda processer och skickar sedan ett meddelande till kernel för att återvinna alla resurser för de terminerade processerna.

2.0.3 InterProcess Communication

IPC av typ meddelandesändning från punkt till punkt via så kallade sockets. Sockets allokeras dynamiskt och representerar ändpunkter för kommunikation mellan processer. Ett meddelande kan innehålla 1-256 bytes med godtyckliga data.

2.0.4 Start av process

Allmänt:

- Allokera processbeskrivning, stigande nummer + löpnr.
- Kopiera pri, tty, argc, argv, cwd, fdesc[0-2], name från föräldern.
- Initialisera egen termineringssemafor.

- Sätt upp stacken, allokera plats för alla register och lägg upp terminate och processens startadress.

- Länka in processen i någon av activeQ0 - activeQ3 beroende på prioritet.

Start av lättviktsprocess: kopiera lokala fildeskriptorer (pekare) från föräldern, vid close() koontrolleras om anroparen äger motsvarande global fildeskriptor (GFD), i så fall frias GFD:n annars frias bara den lokala fildeskriptorn.

Start av tungviktsprocess: Allokera nya GFD:s, sätt upp lokala fildeskriptorer och kopiera GFD-data från föräldern. Vid dup() ska en ny GFD allokeras och data kopieras från den ursprungliga GFD:n.

2.0.5 Terminering av process

- Sätt T(erminated) -status

- Signalera på termineringssemaforen

- Skicka termineringsmeddelande till kernel (för att fria ev. allokerade resurser)

- Stoppa exekveringen genom att anropa suspend()

Då kernel mottar ett termineringsmeddelande så ska:

- Alla lättviktsprocesser som hör till den aktuella processen termineras.

- Alla öppna filer stängas

- Fria alla minnesblock

- Fria alla sockets och IPC-buffrar

- Sätta F(ree) -status

- Sätta processid. = -1

2.0.6 Remote Procedure Call

Alla server-baserade systemanrop utförs med hjälp av ett standardiserat RPC-format där meddelanden skickas med de inbyggda IPC-primitiverna.

Fält	Offset	Antal bytes	Beskrivning
net_src	0	2	Avsändarens nätverksadress

Tabell 1. *Format på RPC-meddelanden.*

Fält	Offset	Antal bytes	Beskrivning
net_dst	2	2	Mottagarens nätverksadress
net_size	4	1	Storlek på användardata
protocol	5	1	Protokoll (0=user, 1=RPC)
module	6	1	Server modul-id.
service	7	1	Servicekod inom aktuell modul
seq	8	2	16 bits sekvensnummer
result	10	2	Resultatkod (0=ok)
header	12	0-244	Användardata, fix del
data	?	0-244	Användardata, variabel del

Tabell 1. *Format på RPC-meddelanden.*

2.0.7 Standard RPC

Följande RPC-anrop bör hanteras av alla typer av server-processer och kan användas för generell övervakning och mätning i systemet.

Service	Procedur	Parametra r	Resultat	Beskrivning
0	null	-	-	Utför ingenting, returnerar alltid result = 0.
1	ver	-	data(0), data(1), data(2-65)	Returnera versionsnummer i data(0-1), LSB på index 0 och en versionssträng i data(2-65). Strängens längd anges av data(2).
2	shutdown	-	-	Terminera servern.
3	restart	-	-	Starta om servern.

Tabell 2. *Standardiserade RPC-meddelanden till alla servers.*

3.0 Filsystem

3.0.1 Inledning

Anropsgränssnittet liknar UNIX; varje process har en liten tabell (< 16) med filbeskrivningar. När en fil öppnas så returneras ett index i processens filtabell vilket sedan används för att referera till den öppna filen. Möjlighet finns att läsa data från en fil blockerande eller icke-blockerande. Hårdvaruenheter hanteras som speciella filer av teckentyp eller blocktyp (terminaler resp. diskar). Enheter och vanliga filer hanteras på samma sätt genom att använda open, read, write mm. Under ytan döljer sig ett distribuerat filsystem som bygger på RPC mellan klienter och speciella server-processer som hanterar den direkta manipulationen av filer och enheter. Protokollen inom filsystemet bygger på principen med tillståndslösa servers, dvs varje RPC innehåller all information som behövs för att utföra den önskade operationen och hanteras helt oberoende av föregående anrop.

3.0.2 Universal File Identifiers

Så kallade UFIDs används som lägesoberoende identifierare för filer och enheter i det distribuerade filsystemet. En UFID innehåller fullständig information om en fil så att den kan refereras oberoende av läge i ett nätverk. Filer kan dock ej flyttas mellan olika maskiner/servers utan att modifiera motsvarande UFID. Detta beror på att en UFID innehåller nätverksadressen till den server som hanterar det filsystem (disk) som filen ligger på. Tabellen nedan visar det exakta formatet på en UFID.

Offset	Storlek	Fältnamn	Beskrivning
0	2	server	Server-adress, MSB = maskin, LSB = socket.
2	1	dev	Filtyp och enhet, 4 msb = filtyp, 4 lsb = enhetsnr.
3	1	spare	Reserverad för framtida bruk.
4	2	id	Lokal filidentifierare, 4 msb = löpnr, 12 lsb = nodnr. eller liknande.

Tabell 3. *Struktur på Universal File IDentifier.*

Fältet “dev” i en UFID anger vilken typ av fil som refereras och vilken subenhet/filsystem som filen ligger på. En server kan hantera upp till 16 subenheter. Det finns i huvudsak tre typer av filer: vanliga, blockorienterade och teckenorienterade. En vanlig fil lagras oftast på någon disk och ett godtyckligt antal tecken kan både läsas och skrivas på godtycklig position i filen. Blockorienterade filer representerar enheter där endast datablock av en fix storlek kan läsas eller skrivas. Teckenorienterade filer representerar enheter där data läses och skrivs tecken för tecken, t ex terminaler och kommunikationsportar. Tabellen nedan visar vilka filtyper som finns och hur de kodas.

Kod	Namn	Beskrivning
0	ftyp_reg	Vanlig fil, både sekvensiell och direkt access kan utföras.
1	ftyp_dir	Directory-fil, endast läsning tillåten, speciella operationer finns t ex rename(), delete() mm..
2	ftyp_stream	Teckenorienterad enhet, endast sekvensiell läsning/skrivning.
3	ftyp_block	Blockorienterad enhet, endast läsning/skrivning av block med 1024 bytes.

Tabell 4. *Kodning av filtyper i UFID.*

3.0.3 Directory server

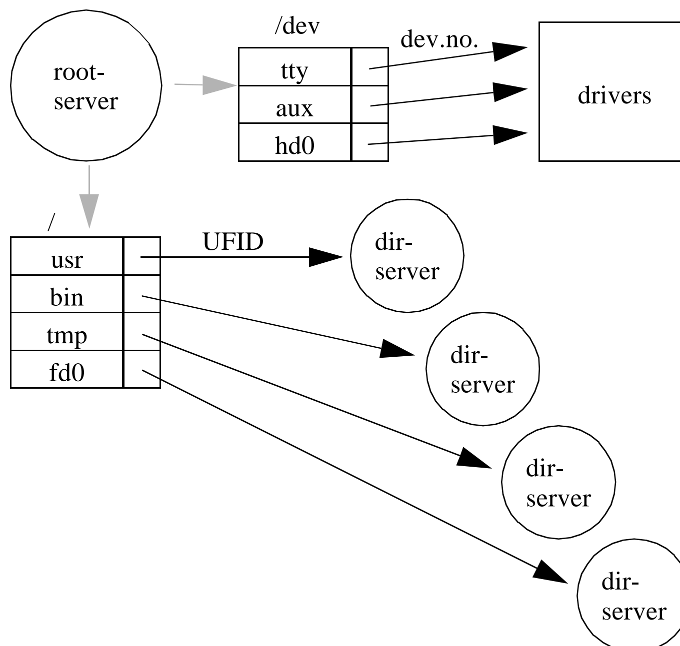
Följande tjänster erbjuds av en directory server:

- lookup(path,mode) => UFID
- rename(path,new)
- delete(path)
- create(path) => UFID
- createDir(path) => UFID
- read_dir(UFID,index,size) => lista med matchande filnamn

3.0.4 File server

- read(UFID,pos,size) => data, size
- write(UFID,pos,size,data) => size
- getAttr(UFID) => filattribut (ägare, accessrättigheter, tid, datum mm)
- setAttr(UFID,filattribut)

3.0.5 Systemstruktur



Root-servern hanterar ett virtuellt filsystem där directory-namn mappas till UFIDs för riktiga filsystem. Alla riktiga filsystem hanteras av någon directory-server. När ett nytt filsystem blir tillgängligt så monteras det dynamiskt i rotkatalogen genom att det nya systemets UFID associeras med ett namn hos root-servern. Root-servern hanterar också en virtuell filkatalog (/dev) där enhetsnamn mappas till drivrutiner och där diverse information om varje enhet lagras.

4.0 Memory server

4.1 Funktion

En memory server administrerar allt dynamiskt allokerbart minne i systemet. Det finns bara en memory server på varje maskin och denna är en del av kernel-processen.

4.2 RPC-gränssnitt

Gränssnittet mot en memory server bygger på standard OS-X RPC. Det exakta meddelandeformatet visas i tabellen nedan.

Fält	Offset	Antal bytes	Beskrivning
RPC header	0	12	Standardiserad RPC-header med avsändare, mottagare, sekvensnummer mm. Se Tabell 1 på sidan 5.
id	12	2	Anroparens process-id.
p	14	2	Pekare till aktuellt minnesblock.
size	16	2	Storlek på aktuellt minnesblock.
map	18	1	Aktuell sidmappning.

Tabell 5. Format på meddelande till memory server.

Alla tjänster som en memory server erbjuder använder det ovan visade meddelandeformatet och för att välja tjänst anges önskad kod i RPC-headern. Parametrar och resultat används och överförs enligt nedan:

Service	Procedur	Parametra r	Resultat	Beskrivning
8	alloc	size, id, map	p	Allokera ett minnesblock i aktuell mappning.
9	galloc	size, id	p, map	Allokera ett minnesblock i godtycklig mappning.
10	release	p, size, id, map		Frigör ett minnesblock.

Tabell 6. *RPC-meddelanden till memory server.*

Service	Procedur	Parametra r	Resultat	Beskrivning
11	clean	id	-	Frigör alla block allokerade av en viss process.
12	change	p, size, id, map	-	Ändra ägare (process-id) för ett visst minnesblock.
13	avail	p, map	p	Returnera antal lediga bytes i viss minnesmappning. Returnerar största lediga block om p <> nil.

Tabell 6. *RPC-meddelanden till memory server.*

5.0 Process server

5.1 Funktion

En process server administrerar start och terminering av processer i systemet. Det finns bara en process server på varje maskin och denna är en del av kernel-processen.

5.2 RPC-gränssnitt

Gränssnittet mot en process server bygger på standard OS-X RPC, detta möjliggör transparent manipulering av processer inom ett nätverk via nätverks-servers. Det exakta meddelandeformatet visas i tabellen nedan.

Fält	Offset	Antal bytes	Beskrivning
RPC header	0	12	Standardiserad RPC-header med avsändare, mottagare, sekvensnummer mm. Se Tabell 1 på sidan 5.
id	12	2	Aktuell process-id.
size	14	2	?
data	16	200	?

Tabell 7. Format på meddelande till process server.

Alla tjänster som en process server erbjuder använder det ovan visade meddelandeformatet och för att välja tjänst anges önskad kod i RPC-headern. Parametrar och resultat används och överförs enligt nedan:

Service	Procedur	Parametra r	Resultat	Beskrivning
8	fork	id	-	Allokera eventuella resurser då en ny process har startats.
9	exit	id	-	Frigör resurser då en process har terminerat.
10	syschk	-	-	Frigör resurser för alla terminerade processer.
11	kill	id	-	Terminera och frigör resurser för en viss process.

Tabell 8. *RPC-meddelanden till process server.*

Service	Procedur	Parametra r	Resultat	Beskrivning
12	pstat	id	size, data	Returnera information om en viss process. OBS id tolkas som index i processtabellen dvs LSB av process-id.

Tabell 8. *RPC-meddelanden till process server.*

6.0 Directory server

6.1 Funktion

En directory server administrerar mappningen från filnamn och sökvägar till filidentifierare (UFIDs) inom ett eller flera filsystem. Olika typer av operationer på sökvägar och läsning av filkataloger mm hanteras också av directory servers.

6.2 RPC-gränssnitt

Gränssnittet mot en directory server bygger på standard OS-X RPC, detta möjliggör transparent sökning efter filer inom ett nätverk via nätverks-servers. Det exakta meddelandeformatet visas i tabellen nedan.

Fält	Offset	Antal bytes	Beskrivning
RPC header	0	12	Standardiserad RPC-header med avsändare, mottagare, sekvensnummer mm. Se Tabell 1 på sidan 5.
uid	12	2	Anroparens användarid.
mode	14	2	Aktuellt arbetsläge / flaggor
index	16	2	Index för läsning av filkatalog
ufid	18	6	Aktuell UFID
path	24	32	Aktuell sökväg / sökmönster / filnamn
data	56	0-128	Variabelt datafält, används vid läsning av filkataloger mm.

Tabell 9. Format på meddelande till directory server.

Alla tjänster som en directory server erbjuder använder det ovan visade meddelandeformatet och för att välja tjänst anges önskad kod i RPC-headern. Parametrar och resultat används och överförs enligt nedan:

Service	Procedur	Parametra r	Resultat	Beskrivning
8	lookup	path, mode, ufid	ufid	Slå upp UFID för filen enligt sökvägen path, i det filsystem som anges av ufid.

Tabell 10. *RPC-meddelanden till directory server.*

Service	Procedur	Parametra r	Resultat	Beskrivning
9	rename	path, data	-	Byt namn på en fil, namnet anges av path och det nya namnet anges av data.
10	delete	path		Ta bort en fil.
11	create	path, mode, uid	ufid	Skapa en normal fil med angivet namn, user id, file mode.
12	createDir	path, mode, uid	ufid	Skapa en filkatalog med angivet namn, user id, file mode.
13	read_dir	ufid, mode, index	mode, index, data	Läs filkatalog enligt ufid, på angivet index. Katalogen anges av ufid, index är ett löpnummer för läsningen, mode anger hur många bytes som maximalt ska läsas. Vid första anropet ska index=0, vid upprepade anrop ska det index som returnerades vid föregående anrop användas. I fältet mode returneras antalet lästa bytes. Om returnerat index=65535 eller mode=0 så finns inget mer att läsa.

Tabell 10. *RPC-meddelanden till directory server.*

7.0 File server

7.1 Funktion

En file server administrerar mappningen från UFIDs till fysiska enheter eller dataareor i olika filsystem. Den direkta manipulationen av filer och enheter sköts alltid av någon file server. En file server kan hantera upp till 16 olika filsystem med upp till 4096 filer i varje filsystem. Varje filsystem har en lokal 4 bits identifierare och varje fil har en 16 bits identifierare som är unik inom ett filsystem.

7.2 RPC-gränssnitt

Gränssnittet mot en file server bygger på standard OS-X RPC, detta möjliggör transparent access till filer inom ett nätverk via nätverks-servers. Det exakta meddelandeformatet visas i tabellen nedan.

Fält	Offset	Antal bytes	Beskrivning
RPC header	0	12	Standardiserad RPC-header med avsändare, mottagare, sekvensnummer mm. Se Tabell 1 på sidan 5.
uid	12	2	Anroparens användarid.
pos	14	2	Aktuellt position för läsning/skrivning.
posHi	16	2	Aktuell position, mest signifikanta delen.
size	18	2	Antal bytes att läsa/skriva.
ufid	20	6	Aktuell UFID
data	26	0-64	Variabelt datafält, används vid överföring av data och filattribut mm.

Tabell 11. Format på meddelande till file server.

Alla tjänster som en file server erbjuder använder det ovan visade meddelandeformatet och för att välja tjänst anges önskad kod i RPC-headern. Parametrar och resultat används och överförs enligt nedan:

Service	Procedur	Parametra r	Resultat	Beskrivning
8	read	uid, pos, posHi, size, ufid	size, data	Läs ett datablock på angiven position i angiven fil, returnerar verkligt antal lästa tecken och det lästa datablocket.
9	write	uid, pos, posHi, size, ufid, data	size	Skriv ett datablock på angiven position i angiven fil, returnerar verkligt antal skrivna tecken.
10	getAttr	ufid	data	Hämta alla filattribut för angiven fil.
11	setAttr	ufid, data	-	Sätt alla filattribut för angiven fil.
12	ioctl	ufid, size, data	size, data	Utför enhetsspecifik operation på angiven fil, både läs/skriv.
13	getStat	-	size, data	Hämta statusinformation om alla filsystem och servern.

Tabell 12. *RPC-meddelanden till file server.*

8.0 Exec server

8.1 Funktion

En exec server hanterar exekvering av programfiler, dvs start av tungviktsprocesser.

8.2 RPC-gränssnitt

Gränssnittet mot en exec server bygger på standard OS-X RPC, detta möjliggör transparent exekvering av programfiler lokalt eller på valfri maskin i ett nätverk. Det exakta meddelandeformatet visas i tabellen nedan.

Fält	Offset	Antal bytes	Beskrivning
RPC header	0	12	Standardiserad RPC-header med avsändare, mottagare, sekvensnummer mm. Se Tabell 1 på sidan 5.
uid	12	2	Anroparens användarid.
stdin	14	6	UFID för stdandard infil.
stdout	20	6	UFID för standard utfil.
stderr	26	6	UFID för standard felutskriftsfil.
argc	32	2	Antal argumentsträngar i argv.
argv	34	0-200	Variabel argumentarea, innehåller packade argumentsträngar där första tecknet i varje sträng anger dess längd.

Tabell 13. Format på meddelande till exec server.

Alla tjänster som en exec server erbjuder använder det ovan visade meddelandeformatet och för att välja tjänst anges önskad kod i RPC-headern. Parametrar och resultat används och överförs enligt nedan:

Service	Procedur	Parametra r	Resultat	Beskrivning
8	exec	uid, stdin, stdout, stderr, argc, argv	uid	Ladda in programfilen enligt första strängen i argv och sätt upp fildeskriptor 0-2 enligt stdin, stdout, stderr. Starta programmet som en tungviktsprocess med argument enligt argc & argv. Returnerar den nya processens id. i uid.

Tabell 14. *RPC-meddelanden till exec server.*

9.0 Name server

9.1 Funktion

Name servern är en allmän nätverkstjänst för dynamisk bindning av namn till nätverksadresser. För att vara allmänt åtkomlig så använder name servern en reserverad socket (id 2). Name servern administrerar en databas som är indelad i ett antal domäner. Varje domän innehåller en tabell med bindningar av nyckelsträngar till heltal. För att slå upp en bindning så utförs ett RPC till name servern med parametrarna: domännamn och nyckelsträng, som resultat fås en felkod och eventuellt det till nyckelsträngen bundna heltalet. Förutom att söka det heltal som är bundet till en viss nyckel så är det möjligt att söka upp den nyckel som är bunden till ett visst heltal i en given domän. Name servern tillhandahåller också funktioner för att läsa av samtliga domännamn och hela domäner.

9.2 RPC-gränssnitt

Gränssnittet mot name servern bygger på standard OS-X RPC, detta möjliggör transparent sökning efter tjänster inom ett nätverk via nätverks-servers. Det exakta meddelandeformatet visas i tabellen nedan.

Fält	Offset	Antal bytes	Beskrivning
RPC header	0	12	Standardiserad RPC-header med avsändare, mottagare, sekvensnummer mm. Se Tabell 1 på sidan 5.
domain	12	16	Aktuellt domännamn
key	28	16	Aktuell nyckelsträng
value	44	2	Aktuellt värde
data	46	0-128	Variabelt datafält, används vid läsning av hela tabeller.

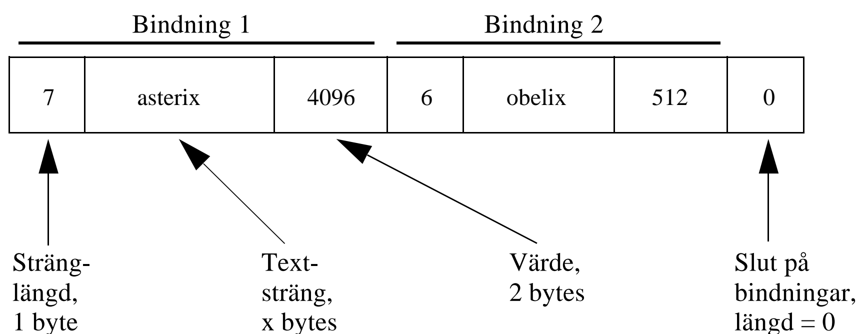
Tabell 15. Format på meddelande till name-server.

Alla tjänster som name servern erbjuder använder det ovan visade meddelandeformatet och för att välja tjänst anges önskad kod i RPC-headern. Parametrar och resultat används och överförs enligt nedan:

Service	Procedur	Parametra r	Resultat	Beskrivning
8	lookup	domain, key	value	Sök värde för domain, key. Returnera value = aktuell bindning, result = 0 om bindning fanns.
9	get_doma ins	value	value, data	Läs domännamn på index enligt value, returnera noll eller flera namn/värde-par i data och nästa index i value. Result = 0 om giltigt index, returvärdet i value < 65535 om fler bindningar finns och värdet ska användas som index i nästa läsning.
10	get_keys	domain, value	value, data	Läs tabellrad på index enligt value och domain, returnera noll eller flera namn/värde-par i data och nästa index i value. Se get_domains ovan.
11	bind	domain, key, value		Skapa bindningen domain, key, value, returnera error = 0 om operationen lyckades.
12	unbind	domain, key		Ta bort bindningen för domain, key. Error = 0 om operationen lyckades, dvs även om bindningen saknades.

Tabell 16. *RPC-meddelanden till name server.*

Vid anrop av get_domains eller get_keys så returneras noll eller flera bindningar ihop packade i fältet data enligt nedan:



10.0 Network server

10.1 Funktion

En network server klarar av full duplex kommunikation över en dubbelriktad länk, men protokollet kräver att maximalt ett sänt obekräftat paket får finnas. Detta gör att endast ett paket behöver buffras för sändning och ett för mottagning. En strikt flödeskontroll fås pga att varje sänt paket måste bekräftas innan nästa kan sändas. Varje paket med data märks av sändaren med ett sekvensnummer. Detta nummer är ett löpnummer i intervallet 0-127. När ett datapaket tas emot så kontrolleras om dess sekvensnummer är lika med det senast mottagna sekvensnumret, i sådana fall är paketet ett duplikat och ingoreras. Dock skickas för varje mottaget paket med korrekt format och checksumma, ett bekräftelsepaket över länken, dvs även ett duplikat bekräftas. När ett mottaget paket accepterats så sparas dess sekvensnummer för kontroll av nästa paket. Ett bekräftelsepaket saknar datadel och har sekvensnummer 128 för att kunna särskiljas från ett datapaket.

Paket med maximalt 250 bytes användardata skickas över en eller flera datalänkar. Data skickas till och från länk-servern med IPC, den första delen av meddelandehuvudet enligt OSX RPC används av datalänk-servern för adressering. När länk-servern startas så anses länken vara nere och då skickas paket innehållande lokalt nodnamn och adress ut på länken med jämna mellanrum tills svar fås. Svaret innehåller andra sidans nodnamn och adress och denna information skickas till den lokala name-servern. Den andra sidans adress sparas av länk-servern och används för att styra utgående paket till rätt datalänk. En tabell med kopplingar mellan destinationsadresser och fysiska länkar administreras av länk-servern.

Om ett testpaket kommer in på någon länk så svarar servern med sitt eget nodnamn och adress, även om länken redan är uppe. När ett testpaket eller ett svar på ett testpaket mottagits så antas länken vara uppe. Sekvensnummren i testpaketen används för att synkronisera de bägge sidornas sekvensnummerräknare. Om ett RPC-paket ska förmedlas via en länk som är nere så svarar länk-servern själv med en negativ ack (result=-1). Efter 5 (?) misslyckade omsändningar så anses länken vara nere och den lokala name-servern informeras då om vilken destination som ej längre är uppe.

10.2 RPC-gränssnitt

Normalt så används network servern för paketförmedling via ett nätverk. Data skickas då med primitiverna för IPC till serverns socket. Network servern använder en reserverad socket (id 0) för att vara allmänt åtkomlig. För att kunna hantera IPC via nätverk så kräver servern att alla förmedlade meddelanden har nedanstående format:

Fält	Offset	Antal bytes	Beskrivning
net_src	0	2	Avsändarens socket id
net_dst	2	2	Mottagarens nätverksadress
net_size	4	1	Storlek på användardata
protocol	5	1	Använt protokoll (0=raw, 1=RPC)
user_data	5	< 250	Användardata

Tabell 17. *Format på IPC paket till network server.*

10.3 Länkprotokoll

För att hantera nätverk av maskiner används en process som hanterar nätet på länknivå. På datalänknivå används ett enkelt protokoll som säkerställer att data överförs korrekt mellan två maskiner. Protokollet använder checksummor, bekräftelser och ett enkelt sekvensnummersystem, vilket garanterar att ett meddelande överförs maximalt en gång.

Tabellen nedan visar formatet på de paket som skickas över kommunikationslänkar mellan network servers:

Fält	Offset	Antal bytes	Beskrivning
start	0	1	Starttecken, alltid = 255
check	1	1	Modulo 256 checksumma på byte 3-9
datchk	2	1	Modulo 256 checksumma på data
status	3	1	Bit 7 = ack, bit 0-6 är sekvensnummer
source	4	2	Avsändarens socket id
dest	6	2	Mottagarens socket id
size	8	1	Antal bytes i datadelen
protocol	9	1	Använt protokoll
data	10	0-250	användardata

Tabell 18. *Paketformat på datalänken.*

10.4 Tillståndsbeskrivning

En network server implementeras lämpligen som en tillståndsautomat. Automaten kan befinna sig i något av dessa tre tillstånd:

Idle: ingen trafik åt något håll
Receive: paket under mottagning från länken
Sent: paket skickat på länken men ej bekräftat

Här följer en kort beskrivning i pseudokod.av hur de olika tillstånden hänger ihop.

Tillstånd idle

```
Om data finns i mottagningsbufferten:
        nollställ mottagningsräknaren och byt tillstånd till receive.
Om meddelande väntar på lokal socket:
        hämta meddelandet, skapa IP-paket, skicka på länken och byt tillstånd
till sent.
Annars:
        frigör processorn genom att vänta en stund.
```

Tillstånd receive

```
Om data finns i mottagningsbufferten:
        hämta tecken
        Om mottagningsräknaren är noll:
                   Om tecknet = starttecknet: spara i paketbufferten och räkna
upp  mottagningsräknaren
                   Annars: fel, byt tillstånd till idle.
        Om mottagningsräknaren = storleken på pakethuvud:
                   kontrollera checksumman
                   Om korrekt: ta emot datadelen i paketet.
                   Annars: fel, vänta en stund, töm  mottagningsbufferten och gå
till                              tillstånd idle.
        Om mottagningsräknaren = storleken på hela paketet:
                   kontrollera checksumman på datadelen och sekvensnumret
                   Om korrekt: skicka bekräftelsepaket, skicka det mottagna
paketet till lokal socket och gå till tillstånd idle.
                   Om fel sekvensnummer: skicka bekräftelsepaket, ignorera paketet
och gå till tillstånd idle.
                   Annars: fel, ignorera paketet och gå till tillstånd idle.
Om mottagningsbufferten är tom:
        kolla om timeout, gå då till tillstånd idle.
```

Tillstånd sent

Tillstånd sent hanteras på samma sätt som receive förutom att om mottagningsfel i tillstånd sent uppstår så ska samma tillstånd behållas, dvs ingen övergång till ilde får ske.

11.0 Drivrutiner

11.1 Inledning

OSX kan delas in i tre lager eller nivåer, där varje lager innehåller kod på en viss abstraktionsnivå relativt hårdvaran:

1. Hårdvarudefinitioner och accessprocedurer
2. Hårdvaruberoende drivrutiner
3. Systemprocesser och hårdvaruoberoende kod

Mjukvaran är nedbruten i ett antal olika kompileringsenheter och endast två (?) av dessa är kopplade till hårdvaran. Övriga moduler har endast länkberoenden till de hårdvaruberoende modulerna. Genom denna uppdelning fås att endast 10% (?) av koden måste modifieras och kompileras om för att flytta operativsystemet till en ny hårvara, dvs 90% av systemet behöver ej kompileras om, endast en omlänkning krävs. För att uppnå denna höga andel maskinoberoende kod så har ett internt anropsgränssnitt definierats mot de hårvaruberoende modulerna.

11.2 Gränssnitt mot hårdvaran

null_int;
Generell avbrottshanterare, utför endast RETI.

set_int(vector: int; handler: pointer);
Sätter upp en avbrottshanterare för angiven avbrottsvektor.

set_map;
Maskinkodsrutin, sätter upp minnesmappning enligt A-registret.

reset_watchdog;
Maskinkodsrutin, återställer watchdog-kretsen. Denna rutin måste anropas minst 10 gånger per sekund.

set_timer(id, time: int; handler: pointer);
Sätter upp angiven hårdvarutimer enligt time, och installerar den angivna avbrottshanteraren för denna timer. OSX kräver att timer 0 är en 16bits timer och att timer 1 och 2 är 8bits timers.

get_driver(id: int; buf: hw_info_ptr): boolean;
Hämtar information om angiven drivrutin och returnerar sant om drivrutinnummret var giltigt. Informationen kopieras till dataarean som utpekas av parametern buf. Följande information fås: enhetsnamn, typ, anropsadresser för dev_read, dev_write, dev_ioctl, hårdvarans basadress och första avbrottsvektorn.

get_sysinfo(item: int): int;
Returnerar information om systemets hårdvarukonfiguration, t ex klockfrekvens CPU-typ, minnesareor för olika mappningar, antal mappningar mm.

init_hardware();
Initialiserar hårdvara, avbrottsvektorer mm.

get_hostname(): ^string;
Returnerar maskinens nodnamn.

set_hostname(s: string);
Sätter om maskinens nodnamn.

get_hostaddr(): int;
Returnerar maskinens nodadress.

set_hostaddr(n: int);
Sätter om maskinens nodadress.

11.3 Gränssnitt mot drivrutiner

dev_read(gfd,buf: pointer; size: int): int;

dev_write(gfd,buf: pointer; size: int): int;

dev_ioctl(gfd,buf: pointer; cmd,size: int): int;

12.0 Systemanrop

12.0.1 Minneshantering

mem_alloc(size: int): pointer;
Allokerar ett minnesblock i den aktuella mappningen och returnerar blockets startadress eller nil om blocket ej kunde allokeras.

mem_release(p: pointer; size: int);
Återlämnar ett minnesblock i den aktuella mappningen.

gmem_chown(p: pointer; size,id: int; map: char);
Ändrar ägare för minnesblocket som anges av p, size och map. Parametern id anger processid. för den nya ägaren av minnesblocket.

gmem_avail(map: char): int;
Returnerar det totala antalet bytes med ledigt minne i angiven mappning.

gmem_max(map: char): int;
Returnerar antalet bytes i det största sammanhängande blocket med ledigt minne i angiven mappning.

mem_avail(): int;
Returnerar det totala antalet bytes med ledigt minne i den anropande processens mappning.

mem_max(): int;
Returnerar antalet bytes i det största sammanhängande blocket med ledigt minne i den anropande processens mappning.

12.0.2 Processhantering

lock();
Förhindrar processbyte och interupt.

unlock();
Tillåter processbyte och interupt.

await(var q: waitq);
Blockerar den anropande processen och placerar den sist i kön q.

cause(var q: waitq);
Aktiverar den första processen i kön q. Om kön är tom händer ingenting.

_cause;
Maskinkodsversion av cause(), parametern överförs i register IX.

_await;
Maskinkodsversion av await(), parametern överförs i register IX.

_icause;
Specialversion av _cause() där aktivering av högprioriterad process sker direkt, sk pre-emptive scheduling.

wait(var s: semaphore);
Väntar på semaforen s, om räknaren är noll så blockeras processen, annars räknas räknaren ner.

signal(var s: semaphore);
Signalera på semaforen s, om någon process väntar så aktiveras den, annars räknas räknaren upp.

sys_time(): int;
Returnerar aktuell systemtid, antal millisekunder sedan systemstart modulo 65536.

idSelf(): int;
Returnerar den anropande processens id.

hold(time: int);
Fördröjer den anropande processen i time millisekunder.

sleep();
Utför omedelbart processbyte.

exit(status: int);
Terminerar den anropande processen med angiven statuskod.

stop();
Stoppar den anropande processen utan att terminera den.

join(id: int): int;
Blockerar den anropande processen tills processen id terminerar eller stoppas, returnerar sedan processens exit-status.
Om processen id ej existerar så returneras -1.

kill(id: int): int;
Terminera processen id omedelbart, returnerar 0 om operationen lyckats. Om processen saknas returneras -1.

_tty_kill;
Maskinkodsfunktion som terminerar alla processer med break-flaggan satt.

set_priority(id,pri: int);
Sätt prioritetsnivån för processen id. Prioritet 0 är lägst och 3 högst.

set_break(id: int; on: boolean);
Sätt break-flaggan för processen id enligt on. Om break-flaggan är sann så termineras processen av _tty_kill().

pstat(id: int; buf: ^pstatbuf): boolean;
Hämtar information om processen id och lagrar den i dataarean som utpekas av variabeln buf, returnerar true om den angivna processen existerar.

argv(n: int): ^string;
Returnerar en pekare till den anropande processens argument-sträng nummer n.

argc(): int;
Returnerar antalet argument i den anropande processens argumentlista.

argp(): pointer;
Returnerar den anropande processens argumentpekare, dvs den pekare som skickats till fork då processen startades. Denna pekare har inget att göra med argumentlistan till processen/programmet.

fork(proc: procedure; stack_size: int; arg: pointer; name: string): int;
Skapar och startar en lättviktsprocess, den nya processen ärver förälderns standardfiler, argumentlista mm. Parametern stack_size anger storleken på den nya processens stack, om stack_size = 0 så får den nya processen samma stackstorlek som föräldern. Parametern arg är en godtycklig pekare som den nya processen kan hämta genom att anropa argp() enligt ovan. Parametern name anger den nya processens namn. Om fork lyckas så returneras den nya processens id, annars -1.

init_waitq(var q: waitq);
Initialiserar processkön q.

init_semaphore(var s: semaphore; value: int);
Initialiserar semaforen s och sätter dess räknare till value.

12.0.3 IPC

create_socket(): int;
Skapar en kanal för process-kommunikation. Returnerar kanalens id eller -1 om ingen kanal kunde skapas.

alloc_socket(id: int): int;
Allokerar kanalen id om den är ledig, returnerar id om operationen lyckats, annars returneras -1.

destroy_socket(id: int);
Återlämnar kanalen id.

ipc_send(src,dst: int; data: pointer; size: int): int;
Skickar data med storleken size via kanalen src till kanalen dst. Returnerar antalet överförda bytes om överföringen lyckats, annars 0.

ipc_rec(local: int; var src: int; buf: pointer; size: int): int;
Tar emot data med maximal storlek enligt size i bufferten buf via kanalen local. Parametern src sätts till den sändande kanalens id och proceduren returnerar antalet mottagna bytes om överföringen lyckats, annars 0. Om size < 32768 så fås blockerande mottagning, dvs den anropande processen blockeras tills data mottagits. Om size >= 32768 så fås icke-blockerande mottagning av max (size - 32768) bytes. Om inga meddelanden finns vid icke-blockerande mottagning så returneras 0 omedelbart.

ipc_rec_tw(local: int; var src: int; buf: pointer; size,timeout: int): int;
Som ipc_rec() men timeout fås efter angivet antal millisekunder om inget meddelande mottagits innan dess.

rpc_send(src,dst: int; header,data: pointer; hsize,size: int): int;
Som ipc_send() men meddelandet sätts ihop av header och data med angivna storlekar.

rpc_rec(local: int; var src: int; header,data: pointer; hsize,size,timeout: int): int;

Som ipc_rec_tw() men mottaget meddelande delas upp i header och data med angivna storlekar. Om timeout = 0 fås oändlig timeout.

rpc_call(module, service: char; header, buf: pointer; server,hsize,dsize,rsize,timeout: int): int;

Utför ett OS-X standard Remote Procedure Call, dvs nätverkstransparent och med sekvensnummerkontroll av svaret mm. Modul och tjänst anges av module respektive service. Parametern header är en pekare till RPC-meddelandets fixa huvud, storleken av denna del anges av hsize. Parametern buf är en pekare till RPC-meddelandets variabla datadel, storleken av denna del vid sändning anges av dsize och maximal storlek vid mottagning anges av rsize. Parametern server anger socket id. för RPC-anropets destination. Parametern timeout används på samma sätt som i systemanropen rpc_rec och ipc_rec_tw ovan. Returnerar resultatkod från RPC-anropet, eller -1 om meddelandet ej kunde sändas alt. timeout har fåtts.

12.0.4 Filhantering

open(path: string; mode: int): int;

Öppnar filen path enligt mode och returnerar en fildeskriptor (ett litet heltal) om operationen lyckats, annars -1.

close(fd: int);

Stänger filen som är associerad med fildeskriptorn fd.

seek(fd,pos: int);
lseek(fd,posLo,posHi: int);

Flyttar filpositionspekaren för filen fd till angiven position om filen är en vanlig fil, annars har seek/lseek ingen effekt.

fstat(fd: int; buf: ^statbuf): boolean;

Hämtar information om den öppna fildeskriptorn fd och lagrar resultatet i variabeln buf^, returnerar true om operationen lyckats.

ftype(fd: int): char;

Returnerar UFID-typ för den öppna fildeskriptorn fd.

ioctl(fd,cmd: int; arg: pointer; size: int): int;

Utför enhetsspecifikt kommando på den teckenorienterade specialfilen (enheten) som är associerad med fildeskriptorn fd. Returnerar -1 om operationen misslyckats.

dup(fd: int): int;

Duplicerar fildeskriptorn fd och returnerar en ny deskriptor som refererar till samma öppna fil som fd. Returnerar -1 om fd ej var associerad med någon öppen fil eller om den anropande processens deskriptortabell var full.

read(fd: int; buf: pointer; size: int): int;

Läser data av angiven storlek från fildeskriptorn fd och returnerar antalet lästa bytes. Om högsta biten i size är satt så fås icke-blockerande läsning.

write(fd: int; data: pointer; size: int): int;

Skriver data av angiven storlek på fildeskriptorn fd och returnerar antalet skrivna bytes.

read_dir(fd: int; buf: pointer; size: int): int;
Läs data av angiven storlek från fildeskriptorn fd och returnera antalet lästa bytes. Data består av packade strängar där första tecknet i varje sträng anger dess längd. När inga fler filnamn finns fås en sträng med längden noll. Endast hela filnamn läses så antalet lästa bytes kan vara mindre än size även om fler namn finns att läsa. Fildeskriptorn måste vara en öppnad filkatalog, annars returneras noll. Observera att anrop till read_dir() ej kan blandas med anrop till read(), en filkatalog bör endast läsas från början till slut med read_dir().

getc(fd: int): int;
Läser en byte från filen fd, returnerar -1 om filslut.

putc(fd: int; data: char);
Skriver en byte på filen fd.

getchar(): int;
Läser en byte från standard infilen, returnerar -1 om filslut.

putchar(data: char);
Skriver en byte på standard utfilen.

puts(s: string);
Skriver strängen s på standardutfilen.

delete(path: string): int;
Tar bort det angivna filnamnet ur motsvarande filkatalog, om den sista länken till filen tas bort så tas även filens i-nod och eventuella allokerade diskblock bort. Filkataloger kan tas bort endast om de är tomma.

rename(srcpath,dstpath: string): int;
Byter namn från srcpath till dstpath.

chdir(path: string): int;
Sätter den anropande processens arbetskatalog (current working directory) till path.

get_cwd(): ^string;
Returnerar sökvägen till den anropande processens aktuella arbetskatalog.

set_cwd(path: string);
Sätt aktuell arbetskatalog enligt path, ingen felhantering.

mkdir(path: string): int;
Skapar en ny filkatalog med namnet path.

12.0.5 Diverse

get_version(): int;
Returnerar aktuellt versionsnummer på operativsystemet. Om operativsystemet har versionsnummer 2.03 så returnerar get_version heltalet 203.

get_time(): int;
Returnerar aktuell tid som antalet minuter sedan klockan 00:00.

set_time(minute: int);
Sätt aktuell tid till angivet antal minuter. Giltiga värden är 0-1439.

get_date(): int;
Returnera aktuellt datum som antalet dagar sedan år 0, då varje månad antas ha 31 dagar.

set_date(day: int);
Sätt aktuellt datum till angivet antal dagar. Giltiga värden är 0-37199.

exec(host,argc: int; argv: arg_ptr): int;
Startar en ny process på maskinen host, med argc argument enligt argumentlistan argv, genom att ladda in den exekverbara filen vars namn anges av argv^[0]. Stackstorleken för den nya processen läses i programfilens huvud. Om operationen lyckas så returneras den nya processens id. Vid anropet utförs ingen sökning efter filen utan den korrekta sökvägen måste anges i argv^[0]. Vid anropet kopieras den anropande processens fildeskriptorer 0-2 och de angivna argument- och omgivningsareorna till den nya processen. Anropet returnerar så snart som ett programsegment har allokerats och en laddprocess startats, dvs programfilen läses in efter anropet. Eventuella läsfel vid laddningen rapporteras genom att den nya processen terminerar med statuskod skild från noll. Observera att värdmaskinen ska anges som maskinnummer * 256.

12.1 Inhoppsadresser

Samtliga ovan beskrivna systemanrop nås genom anrop till nedan definierade absoluta inhoppsadresser.

Procedur	Adress [hex]
get_sysinfo	FD00
get_driver	FD04
get_hostaddr	FD08
get_hostname	FD0C
set_hostaddr	FD10
set_hostname	FD14
	FD18
	FD1C
mem_alloc	FD20
mem_release	FD24
gmem_chown	FD28
gmem_avail	FD2C

Tabell 19. *Absoluta adresser för systemanrop.*

Procedur	Adress [hex]
gmem_max	FD30
mem_avail	FD34
mem_max	FD38
	FD3C
	FD40
	FD44
	FD48
	FD4C
lock	FD50
unlock	FD54
await	FD58
cause	FD5C
_await	FD60
_cause	FD64
_icause	FD68
wait	FD6C
signal	FD70
sys_time	FD74
idSelf	FD78
hold	FD7C
sleep	FD80
exit	FD84
stop	FD88
join	FD8C
_tty_kill	FD90
set_priority	FD94
set_break	FD98
argv	FD9C
argc	FDA0
argp	FDA4
sys_fork	FDA8

Tabell 19. *Absoluta adresser för systemanrop.*

Procedur	Adress [hex]
fork	FDAC
init_waitq	FDB0
ini_semaphore	FDB4
	FDB8
	FDBC
	FDC0
	FDC4
	FDC8
	FDCC
create_socket	FDD0
alloc_socket	FDD4
destroy_socket	FDD8
ipc_send	FDDC
ipc_rec	FDE0
ipc_rec_tw	FDE4
rpc_send	FDE8
rpc_rec	FDEC
rpc_call	FDF0
	FDF4
	FDF8
	FDFC
open	FE00
close	FE04
seek	FE08
lseek	FE0C
ftype	FE10
dup	FE14
read	FE18
write	FE1C
getc	FE20
putc	FE24

Tabell 19. *Absoluta adresser för systemanrop.*

Procedur	Adress [hex]
getchar	FE28
putchar	FE2C
puts	FE30
	FE34
	FE38
read_dir	FE3C
get_cwd	FE40
set_cwd	FE44
exec	FE48
	FE4C
	FE50
	FE54
	FE58
	FE5C
	FE60
	FE64
	FE68
	FE6C
	FE70
	FE74
	FE78
	FE7C
	FE80
	FE84
	FE88
	FE8C
	FE90
	FE94
	FE98
	FE9C
	FEA0

Tabell 19. *Absoluta adresser för systemanrop.*

Procedur	Adress [hex]
	FEA4
	FEA8
	FEAC
	FEB0
	FEB4
	FEB8
	FEBC
	FEC0
	FEC4
	FEC8
	FECC
	FED0
	FED4
	FED8
	FEDC
	FEE0
	FEE4
	FEE8
	FEEC
	FEF0
	FEF4
	FEF8
	FEFC

Tabell 19. *Absoluta adresser för systemanrop.*