IT 08 037
Examensarbete 30 hp
Oktober 2008

OSE och Linux

En studie om prestanda

Viveka Sjöblom

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

OSE and Linux: A study about performance

Viveka Sjöblom

There are some key elements which distinguish a general operating system, such as Linux, from a Real Time operating system, such as OSE. The later belongs to the category of soft Real Time systems while Linux is a general operating system, with significantly improved real time properties after version 2.6. These improvements make it interesting to compare the two systems.

When comparing two operating systems, there are a number of issues which have to be considered. One way to measure the performance of the systems is by studying the time consumption for the same functionality and hardware.

During the study, the items that appeared as the most important for the performance of the systems, were: process primitives, system primitives, interprocess communication, context switches and interrupt latency.

This work demonstrates that it is possible and reasonable to do performance testing on two different operating systems. The implemented tests cover a number of basic functionalities, which appeared to be the most crucial ones for the performance of the systems.

Handledare: Magnus Karlsson
Ämnesgranskare: Ivan Christoff
Examinator: Anders Jansson
IT 08 037
Sponsor: Enea AB

Tryckt av: Reprocentralen ITC

Innehållsförteckning

1 Bakgrund

Realtidsoperativsystem (RTOS) är vanliga i ett flertal olika områden. Inom inbyggda system är det idag vanligt med RTOS för telefoni, telecom, automatisering, och medicinska tillämpningar. Det speciella med RTOS är att de hanterar tidskritiska applikationer, vilka kräver garantier för att tidsbegränsningar inom systemet inte bryts. I ett RTOS finns möjligheten att garantera exekvering inom en viss tid.

Det finns ett flertal viktiga faktorer som skiljer ett RTOS från ett generellt operativsystem (OS). Ett RTOS har: en strikt prioritetsbaserad schemaläggning, begränsad avbrottslatens och fasta tider för kontextbyten. Vidare är ett RTOS är oftast skalbart, modulerbart och lättviktigt, detta för att möjliggöra avvägningar mellan prestanda och funktion. Förutbestämbarhet är kanske den viktigaste egenskapen hos ett RTOS men som saknas hos generella OS. Utan denna möjlighet till förutbestämbarhet finns inga garantier för att en operation, i alla situationer, utförs inom tidsbegränsningen.

Ett alternativ till RTOS kan vara ett generellt OS som kompletterats med snabbare hårdvara och antagandet att den snabbare hårdvaran räcker till för att möta tidsgränserna för det aktuella systemet. Hårdvaruutvecklingen går mot att inbyggda processorer och mikro controllers blir allt mer kraftfulla och billiga. Sammantaget öppnar detta möjligheterna att billigt producera inbyggda system baserade på generella OS.

Frågan huruvida ett generellt OS duger åt en specifik tidskritisk applikation är komplicerad att svara på eftersom det är så många faktorer som påverkar resultatet. För att underlätta den här typen av designfrågor, bör olika operativsystem vara jämförbara med en objektiv mätmetod. Metoden bör tydligt visa operativsystemets prestanda. En sådan mätmetod skulle kunna vara en uppsättning tester som mäter utvalda nyckelpunkter hos operativsystemet, till exempel tider för avbrottslatens, kontextbyten, signalhantering och footprint.[1]

Syftet med detta magisterprojekt och tillhörande uppsats är att undersöka och utvärdera metoder för analys och verifiering av prestanda hos realtidsoperativsystemet OSE och jämföra detta med det generella operativsystemet Linux. Arbetet består av identifiering av relevant prestanda (vad ska mätas) och utveckling av mätmetoder (hur ska det mätas). Projektet inkluderar även förberedelser till ett ramverk för tester, sådant att testerna enkelt kan upprepas i framtiden för kommande versioner av operativsystemen.[2]

---

1 Denna typ av tester sammanfattas ofta med namnet benchmark-tester.

2 Detta är en förkortad verison av rapporten. För en fullständig rapport och testresultat kan man kontakta mig på viveka.sjoblom@enea.com

1.1 Problemformulering

En benchmark-applikation[3] som ska fungera på två olika operativsystem kommer att kräva en noggrann analys av de båda operativsystemen. Analysen är viktig för att undvika risken att resultaten blir missvisande och det är viktigt att speciell hänsyn tas till skillnader mellan operativsystemen -skillnader som måste reflekteras i testsvitens utformning.
Oavsett om benchmarking-systemet[4] är anpassat för ett, två eller flera operativsystem bör första steget vara att identifiera intressanta tester. Därför kommer det här arbetet att börja med att identifiera tester för prestandamätning av ett realtidsoperativsystem. När detta är gjort måste sedan funktionaliteten undersökas i de båda operativsystemen, för att visa att de utvalda testerna är jämförbara. Evalueringen är ett viktigt steg för att undvika missvisande tester som kan omkullkasta hela tanken med det här projektet. De olika operativsystemen kan ha så olika implementationer av samma funktioner, så det blir omöjligt att göra rättvisa mätningar, även om den aktuella funktionaliteten är ett intressant område för prestandamätning. Exempelvis är det viktigt att undvika jämförelser mellan en lättviktig tråd och en tyngre process.
Evalueringen av de olika operativsystemens funktionalitet sker genom de två systemens manual-sidor samt studier av implementationen i operativsystemskärnan. I OSEs fall med frågor direkt till core-teamets utvecklare.

Nästa steg för att få en bra och rättvis testsvit blir att fundera kring de olika möjligheterna till tidmätning. Problem som kan uppstå kring tidmätning är dels noggrannheten på tidsfunktionen det vill säga i vilken upplösning returneras tiden. De olika systemen har olika möjligheter att mäta tiden och tester får inte formuleras så att ett test tar kortare tid än noggrannheten på tidmätaren. Testerna måste även formuleras så att overhead-tider undviks så mycket som möjligt.

Sista steget för en rättvis jämförelse är att fundera kring kompilering av de två testsviterna, de två testsviterna måste självklart kompileras med motsvarande optimeringar. Utöver test, tidmätning och kompilering så måste sviten exekveras på samma hårdvara. Utan att köra på samma hårdvara är det omöjligt att få en korrekt jämförelse.

1.2 Mål

Målet för detta magisterarbete är att designa och utveckla en uppsättning av benchmark-tester[5] som tillsammans skapar ett generellt ramverk för prestandaanalys mellan OS. Resultatet kommer inte att vara ett heltäckande operativsystemstest utan peka på några nyckelpunkter som är av stor vikt för ett RTOS. Ramverket ska ge möjlighet till en objektiv jämförelse av OSE och Linux. Testerna ska utföras på en lämplig plattform till exempel PowerPc. Och ska utföras på ett sådant sätt som gör det lätt att återupprepa dem på till exempel olika operativsystemsversioner. ·

---

[3] En Benchmark-applikation är en applikation som innehåller ett eller flera prestandatester, som utförs när applikationen körs.

[4] Ett benchmark-system kan bestå av en eller flera benchmark-applikationer men är rent generellt mer komplex än en benchmark-applikation.

[5] Benchmark-tester innefattar tester för prestandamätning av nyckelpunkter för det valda testområdet.

1.3 Slutsatser

Det är möjligt att på ett rättvisande sätt jämföra två olika operativsystem, även om de är av olika typ så som ett generellt OS och ett RTOS. I det här examensarbetet har jämförelse gjorts mellan det generella operativsystemet Linux och realtids operativsystemet OSE. Det är viktigt, för att undvika onödiga skillnader, att välja rätt abstraktionsnivå för testerna. Rätt abstraktionsnivå visade sig finnas bland högnivåapplikationsbenchmarks. Genom att lägga testerna på en systemanropsnivå är det möjligt att testa prestanda, av funktionalitet, på den nivån som en användare kommer att möta systemet. På systemanropsnivå finns den viktigaste funktionaliteten implementerad på de bägge operativsystemen, och därmed ges en möjlighet till jämförelse mellan systemen.

En oväntad svårighet i det här projektet var den bristande noggrannheten för tidmätaren i Linux. Denna gav ingen möjlighet att beräkna tids intervall ned till noggrannheter på klockcykler vilket gjorde det omöjligt att utveckla test för avbrottslatens. Detta problem är dock lätt avhjälpt med en laddmodul som ger rätt funktionalitet, men en sådan fans det ej tid för att utveckla inom ramen av detta examensarbetet.

2 Realtidsoperativsystem

Applikationer är inte bara den typen av program som finns på en vanlig persondator (PC), med en skärm, tangentbord och en hårddisk. Det kan visserligen vara irriterande när filmen eller spelet laggar, men det leder inte till någon katastrof när det inträffar. Men det finns en mängd applikationer i mycket mer känslig miljö än PC-miljön. Det kan gälla till exempel applikationer som styr robotarmar, flygplan, mobiltelefoni, pacemakers eller till exempel bromssystemet i en bil. Den här typen av system måste oftast hantera en mängd olika input samtidigt och dessutom alltid leverera korrekt resultat i tid. Annars kan effekterna bli mycket alvarliga [1].

Det finns många olika definitioner på termen realtid. Eftersom olika system har olika behov uppstår också olika definitioner för att möta de krav som ställs. För somliga system räcker det med en genomsnittlig svarstid[6] medan i andra system måste alla tidsbegränsningar följas [7]. I artikeln RTOS evaluating Project återfinns följande definitioner på realtidssystem.

- *Realtidssystem är system där korrektheten inte enbart bestäms av korrektheten i det logiska resultatet utan även av att resultatet levererades inom tidsbegränsningen.*
- *DIN44300: Modellen för ett realtidssystem är modellen för ett datorsystem i vilket programmen som tar in data utifrån alltid är redo, så att deras resultat är klara inom en förutbestämbar tidsrymd. Indata kommer antingen slumpartat eller vid förutbestämbar tid beroende på applikationen.*

- *Koymans, Kupier, Zijlstra -1988: Ett realtidssystem är ett interaktivt system som upprätthåller en pågående relation med en asynkron miljö. Det vill säga en miljö som fortgår oberoende av realtidssystemet, på ett icke samarbetande sätt.*

- *Realtids (mjukvara) (IEEE 610,12 -1990): tillhör ett system eller modell av operationer i vilken beräkningar utförs under tiden en verklig extern process sker, och på ett sådant sätt att de beräknade resultaten kan användas för att kontrollera och övervaka den externa processen .*

- *Martin Timmerman: Ett realtidssystem svarar på ett tidsmässigt avgörbart sätt på okontrollerbar extern stimuli.*

Det är viktigt att skilja på de två begreppen RTOS och Realtidsystem. Ett RTOS är den grundläggande byggstenen i ett realtidssystem. För att bygga ett förutbestämbart system måste alla komponenter, både hårdvara och mjukvara, göra det möjligt att tidskrav kan sättas och uppfyllas. Om bussen inte håller de utsatta målen spelar det ingen roll hur mjukvaran är konstruerad. Inte heller spelar det någon roll om bussen är otroligt snabb om inte

[6] Svarstiden är det tidsintevall som uppstår mellan det en applikation får ett stimuli tills det appiktionen har producerat ett resultat baserat på det aktuella stimulit.

schemaläggaren ger rätt process tillgång till bussen. Ett bra realtidssystem kan definieras som ett system som har ett förutsägbart beteende under alla systembelastnigssituationer [8].

Slutsats: ett realtidssystem har väl definierade fasta tidsbegränsningar, alla beräkningar måste ske inom de definierade begränsningarna annars fallerar systemet. Realtidssystemen indelas efter hur känsliga de är. Det finns två huvudtyper mjuka och hårda-realtidssystem.

I hårda-realtidssystem finns garantier att alla kritiska uppgifter avslutas i tid. För att uppfylla det här målet måste alla avbrott i systemet vara begränsade, oavsett om det gäller att hämta sparad data eller för tiden det tar OSet att avsluta någon godtycklig uppgift. Hit räknas alla system där mänskligt liv hänger på att systemet levererar i tid eller att en missad deadline innebär katastrofala följder. Exempel på den här typen av realtidssystem är sjukvårdsutrustning och utrustning i bilar som bromsar och airbags.

En mindre begränsad typ är de mjuka-realtidssystemen, här kan det finnas möjlighet att systemet kan återhämta sig efter en missad tidsbegränsning. En kritisk uppgift får prioritet över andra uppgifter och behåller prioriteten tills uppgiften är avslutad. Den här typen av system kräver inte att det är ett RTOS som styr systemet eftersom inte alla avbrott har begränsningar, men det kan vara bra att ett RTOS används som grund. Med tanke på avsaknaden av avbrottsbegränsningar i vissa situationer kan den här typen av system vara ett olämpligt val för en industri med till exempel styrning av robotar. Mjuka-realtidssystem kan emellertid användas till väldigt mycket, de är till exempel väl lämpade för styrning av mobiltelefoner, telefonnät, och hemelektronik [2, 6].

2.1 Nyckelfunktioner i ett realtidsoperativsystem

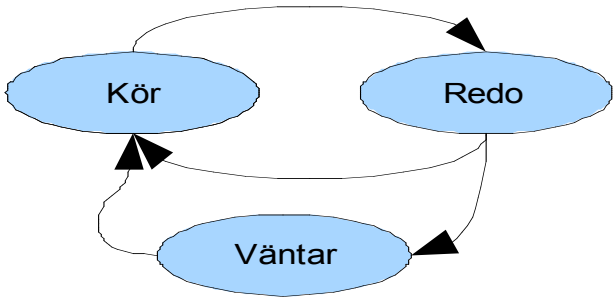
Alla applikationer som utför mer än två uppgifter samtidigt behöver använda sig av multi-tasking. Om applikationen samtidigt tar in tangentbordstryckningar och visar upp dessa som en skriven text, vilket en ordbehandlare gör, behövs multi-tasking och en process/tråd miljö. De flesta applikationer är mycket mer avancerade och består ibland av en blandning med delar av hård-realtid, mjuk-realtid och icke realtid. I dessa situationer blir det självklart att multi-tasking behövs [2].

En multi-tasking miljö är alltså grunden för de flesta applikationer. För en realtidsapplikation gäller även att beteendet måste vara förutsägbart, trots multipla samtidiga externa händelser, som kan ske i en okontrollerad ordning (definitionen av realtid). För mulit-tasking krävs att funktionalitet för trådar och eller processer finns. I UNIX, finns det flera nivåer av den här typen av funktionalitet. Den tyngsta typen är UNIX processer, en äldre typ där kontexten eller miljön för processen är stor. Det leder till att byten mellan processer är tungt och tidskrävande, vilket inte passar för realtidssystem. För att undvika onödigt stora och tidskrävande kontextbyten har därför lättviktsprocesser eller trådar skapats. Lättviktsprocesserna ärver endast en delmängd av föräldraprocessens kontext och blir därmed lättare och snabbare på att utföra kontextbytena och de passar därmed bättre in i en realtidsmiljö.

I en multi-taskingmiljö måste byten från en process till en annan schemaläggas. Om det finns mer än en process måste det finnas någon form av algoritm som kan avgöra vilken process som ska få tillgång till processorn näst. Schemaläggaren har till uppgift att byta från en process till en annan och är en grundläggande del i alla OS. Schemaläggning utgör overhead i systemet, så den måste utföras så snabbt som möjligt.

All schemaläggning sker utifrån processers tre tillstånd, körande, blockerad och redo. En process räknas som körande under den tiden processorn exekverar processens kod. Om processen behöver vänta på systemresurser då är den blockerad, till exempel I/O eller minne. När en blockerad process har fått de resurser den väntade på övergår den till antingen körande eller väntande tillstånd. Väntande tillstånd hamnar den i om det finns någon annan process som är i det körande tillståndet. Eftersom det kan finnas flera processer som väntar på att köra, behövs en kö för väntande processer, denna kö kallas för ”redo-kön”. Om det finns mer än en process i redo-kön så behövs det en beslutsfattande algoritm för att avgöra vilken av processerna som ska få tillgång till processorn.

Figur 1: En process kan vara i något av följande tre lägen, Körande, Väntande, Redo.



De flesta applikationer består av en blandning av realtids- och icke-realtidsdelar. Den del av schemaläggningen där processerna tillhör icke-realtidsdelen, kan schemaläggas enligt någon standard operativsystemsalgoritm, till exempel Round Robin. Oavsätt om det är en mjuk eller en hård-realtid, bör schemaläggaren utformas på ett sådant sätt att realtidsprocesser har den högsta prioriteten. Högre prioriterade processer måste få köra innan lägre, eller icke prioriterade processer får exekvera. Prioritetsnivån får inte heller degradera över tiden.

Antalet prioritetsnivåer behöver vara så pass stort att det räcker för att särskilja de olika processernas behov av att få tillgång till processorn. Icke-realtidsprocesser kan placeras på en lägre eller oprioriterad nivå [8].

När en realtidsprocess med högre prioritet hamnar i redo-kön måste processen med lägre prioritet avbrytas. Tiden det tar innan realtidsprocessen börjar exekvera från det att den hamnat i redo-kön kallas kontextbyteslatens.

Kontext är den information som finns representerad i alla processers Process Control Block (PCB). PCB innehåller information om värdet i CPU register, tillstånd för processer och minneshanteringsinformation. Det som händer vid ett kontextbyte är att en PCB sparas och nästa laddas. Under tiden detta sker kan inte operativsystemet göra något annat. Tiden för kontextbytet är overhead-tid och är därför mycket intressant för ett RTOS. Tiden för kontextbytet varierar bland annat beroende på hårdvaran. Hur snabbt en PCB kan sparas och en annan laddas in påverkas mycket av hur snabbt minnet är.

Kontextbytet måste vara förutbestämbart och självklart utföras så fort som möjligt. Det är oftast tiden det tar att avsluta ett systemanrop som utgör den största delen av kontextbytet. För att begränsa kontextbytet behöver alla systemanrop vara förutbestämbara. Det finns flera sätt att uppnå detta. Ett exempel är att placera ”pre-emption points” i systemanrop som kan dra ut på tiden. Alternativt kan hela operativsystemskärnan göras förutbestämbar [2].

En del i kontextbytet är det avbrott som uppstår till exempel när en prioriterad process hamnar i redo-kön. Avbrott är en viktig del i ett RTOS. Enligt definitionen på RTOS ska ett realtidssystem kunna inom en bestämd tid agera på externa händelser. Detta är viktigt eftersom många realtidsuppgifter är kopplade till periferiutrustningens avbrottssignaler. Det är därför önskvärt att avbrottens servicerutiner ska avslutas så snabbt som möjligt. De förseningar som uppstår i samband med hantering av avbrott kallas avbrottslatens.

Avbrottslatens är alltså den tid det tar från att ett avbrott genererats tills dess exekvering av första instruktionen i avbrottsrutinen påbörjats. Avbrottslatensen kan påverkas av till exempel avbrotts kontrollers, avbrotts maskning och operativsystemets avbrottshanteringsmetoder, med mera.

Problemet med avbrottslatens är att det är svårt att avgöra den maximala tiden det tar för avbrottsrutinen att avslutas och att det (precis som i fallet med kontextbytet) är endast overhead tid i systemet. Normalt körs avbrottsrutinen i ett läge där den inte är mottaglig för avbrott. Om ett högre prioriterat avbrott påbörjas kan detta ta lång tid eftersom det första avbrottet måste slutföras. Det är detta som gör det svårt att beräkna den maximala tiden för avbrotts hanteringen. Genom att köra avbrottsrutinen som en högprioriterad tråd kan den här problematiken lösas [4].

Utöver kontextbyte, avbrottslatens, schemaläggning och processhantering, med mera, finns det ett antal operationer som sker med stor frekvens. Om ett RTOS ska kunna uppfylla kravet på förutbestämbarhet blir det även intressant att studera vanligt förekommande operationer. Till denna grupp räknas att kopiera och flytta små mängder data eller till exempel fylla ett minnesområde med 0 inför någon operation.

Den viktigaste egenskapen i ett RTOS är förutbestämbarhet. Utöver det är följande punkter extra intressanta och därför viktiga att testa.

- Multi-tasking
  Trådar/processer. Dessa måste vara lätta så byten inte blir onödigt tunga och därmed tidskrävande.
- Schemaläggning
  Måste ta hänsyn till prioritet och ske så snabbt som möjligt.
- Prioritet på processer
- Begränsad avbrottslatens
- Kontextbyte
- Minneshantering

2.2 Linux är inget realtidsoperativsystem

Operativsystemet Linux ökar i popularitet men det är långt ifrån ett RTOS. Det kan inte räknas till gruppen RTOS eftersom det bland annat har varierande avbrottslatens. Det finns mycket forskning kring att förbättra Linuxkärnas realtidsegenskaper. Från och med version 2.6 har kärnan inbyggd förutbestämbarhet men det räcker inte för att Linux ska kunna räknas som ett RTOS, eftersom avbrottslatensen fortfarande är osäker [4, 5].

För att en operativ miljö[7] ska tillmötesgå hårda realtidskrav måste miljön garantera att alla av en applikations deadlines hålls. I mjuka-realtidssystem finns inga garantier och deadlines kan missas. Det gör att i praktiken kan ett generellt OS så som Linux erbjuda tillräcklig säkerhet för en applikation med relativt långa deadlines och av typen mjuk-realtid.

Eftersom generella OS inte lämnar garantier för att deadlines hålls bör noggrann testning utföras innan den här typen av OS används i ett realtidssystem. Trots noggrann testning kan det finnas situationer, som inte klara deadlines, och som är svåra att provocera fram. Dessa situationer kan orsakas av till exempel att det finns algoritmer som inte har deterministisk prestanda någonstans i koden.

Det finns många kvalitéer som är viktiga för att ett OS ska vara lämplig för att köra realtidsapplikationer. OSet behöver vara multi-trådat, förutbestämbart och ge möjlighet till olika prioritetsnivåer. I Linux behöver realtidsapplikationer som körs i user-mode använda funktionalitet så som schemaläggning, interprocess kommunikation och en mängd systemanrop.

[7] Operativ miljö innerfattar både OS och alla körande processer, avbrott och hårdvaruaktiviteter.

I ref [7]nämns följande systemanrop som extra viktiga i realtidssammanhang ”*exit(2), fork(2), exec(2), kill(2), pipe(2), brk(2), getusuage(2), mmap(2), settimer(2), ipc(2) (med hjälp av semget(), shmget(), msgget(), clone() och setsched())*”.

Det finns kommersiella företag, till exempel Monta Vista, som erbjuder realtids-patchar till Linux. Det finns flera metoder att förbättra realtidsegenskaperna hos Linux system. Generellt kan metoderna delas in i två kategorier, hårdvaru- och mjukvaruförbättringar. Metoden för hårdvara innebär att utnyttja snabbare processorer och mer avancerad hårdvara. Vilket inte leder till större förutbestämbarhet. Mjukvarumetoden försöker förbättra realtids egenskaperna i kärnan. Det finns flera realtids Linux releaser baserade på både kommersiella och open source distubutioner. I ref [4] finns en mängd realtids Linux Distributioner nämnda där ibland följande.

- RT Linux Pro erbjuder en POSIX hård realtidsmiljö som kör Linux som en sovande tråd. Realtidsapplikationerna körs huvudsakligen inom hårdvarubegränsningarna. Resterande Linux funktionalitet är endast tillgänglig som en integrations plattform.
- Klinux Koan software Engineeing finns i Italien. De säljer och supportar realtids Linux och utvecklingsverktyg för industriella applikationer, baserat på RTAI realtids extensions.
- µLinux Lineo Sulutions (Japan) är en spinn-off från företaget Lineo/Embedix Company (US). Med väldigt liten footprint, korta boot- och avslutningstider samt support för hård-realtid i Linux kernel-space. Lineo Sulutions fokuserar på elektroniska konsumentprodukter.
- LynuxWorks: BlueCat RT garanterar tidskritisk hantering av interrupt och andra lågnivå hårdvaruoperationer genom att implementera en fullständig Linuxkärna som en tråd som tillhör ett mindre och mycket responskänsligt RTOS. Detta minimala RTOS fångar interrupt och andra lågnivå funktioner och gör en preliminär processning innan avbrottet skickas vidare till Linux tråden.
- Monta Vista Software: Realtidslösningar för Linux. Monta Vistas Realtids Linux lösning är baserad på att addera speciell funktionalitet till deras robusta Linuxkärna. Funktionaliteten består av: förutbestämbar Linuxkärna, realtids-schemaläggare, högupplösta POSIX timers och ett ”high avabillity framework ”.
- RedHawk Concurrent Computer Corp. Är i huvudsak en hårdvaruleverantör som även marknadsför RedHawk, ett Linux RTOS baserat på en kärna för multiprocessorena Xeon och Opteron systems.
- TimeSys, REDSonic, med flera.

För att avgöra om en Linux distribution är lämpad för att användas i realtidssammanhang bör man studera samma nyckelfunktionalitet som är viktig i ett RTOS (mer om detta återfinns i 2.1) De huvudgrupper som är viktigast är avbrottslatens, tiden för kontextbyte och schemaläggningslatens, IPC och processhantering. Utöver dessa finns det ett antal systemanrop som hanterar minne. Några är mycket frekvent förekommande exempel på dessa är *memset(), memmov()* och *memcpy().*

Eftersom det ovan är visat att Linux i går att använda i ett mjukt realtidssystem så blir målet med detta projekt blir att utveckla prestanda tester för operativsystem. Testerna ska beröra de områden, som tidigare beskrivits som viktiga, och samtidigt möjliggöra en rättvis jämförelse mellan OSE5 och realtids Linux.

2.3 Överblick av några utvalda benchmarksystem

Med *benchmark* menas den aktivitet som syftar till att visa relativa prestanda hos ett objekt. Den relativa prestanda är i datorsammanhang resultatet av ett antal utförda standardtester på, till exempel en operation, ett program eller en grupp av program. Benchmarks erbjuder alltså en metod för att jämföra prestanda på olika subsystem oberoende av plattform eller arkitektur.

Behovet av benchmark har vuxit fram i samband med att datorarkitekturen utvecklats. När det inte längre var möjligt att jämföra prestanda endast genom att studera specifikationer började olika typer av benchmarks att utvecklats [12].

Några vanliga typer är ”syntetiska” och ”applikations” benchmark. Med syntetiska Benchmark menas tester som är utformade för att mäta prestanda på en individuell komponent i ett system. Ofta genomförs detta genom att belasta komponenten maximalt. Ett välkänt exempel på ett syntetiskt benchmark är Whetstone skapad av Harold Curnow 1972. Enligt André D. Basala, ref [10] är denna testsvit fortfarande vanligt förekommande. Whetstone mäter effektivitet på flyttalsberäkningar i en CPU.

Det huvudsakliga problemet med syntetiska benchmark-tester är att de inte representerar systemets prestanda i en verklig situation. Syntetiska benchmark-tester beskriver i bästa fall en specifik aspekt av det testade systemet. Det vill säga att syntetiska Benchmarks är bra så länge det finns förståelse för uppgiften och dess begränsningar. Alternativet till syntetiska benchmarks är applikations-benchmarks. Applikations-benchmarks bygger på att utnyttja en verklig applikation och testa prestanda med hjälp av den. Exempel på den här typen av Benchmarks är Linux Test Project (LTP) ref [15] som antagligen är det största benchmark projektet och även det nyaste då en version släpptes under detta arbetes gång.

Syntetiska och applikations-benchmarks kan delas in i lågnivå- och högnivå-benchmarks. Lågnivå-benchmarks mäter hårdvarans prestanda, till exempel CPU klockan, minnes-accesstider med mera. Lågnivå-benchmarks kan användas till att testa om en drivrutin fungerar korrekt tillsammans med en enhet.

Högnivå-benchmarks är mer fokuserade på prestandamätning på kombinationen hårdvara, drivrutiner, OS och applikationer för en specifik aspekt av ett system. Typiska högnivåtester är filsystemets I/O prestanda eller systemanrop som tillexempel *fork()*. Högnivå benchmark kan vara antingen syntetiska eller applikations-benchmark. Självklart är alla lågnivå-benchmark syntetiska [10].

Det som behövs för att kunna jämföra ett RTOS med ett generellt OS är högnivå applikations-benchmark. Detta eftersom denna typ säger mer om systemets verkliga prestanda samt att det

gör det möjligt att bortse från hur till exempel drivrutiner är implementerade. (Det finns en mängd olika benchmark system som mäter olika funktionalitet.)

Eftersom OSE är det RTOS som är intressant i detta fall, så är benchmarks som fungerar tillsammans med OSE intressanta att studera. I OSE finns det en uppsättning tester som går under benämningen P-Bench. P-Bench är ett högnivå applikations benchmarksystem som testar de flesta av de intressanta operationerna i ett RTOS. Några exempel på tester är *alloc()*, *free()*, *send(), recieve(), create(), kill()* och tester som send-swap som mäter schemaläggning. Eftersom P-Bench är utvecklat för OSE är det inte möjligt att köra det på någon Linux distribution.

För generella OS finns det en uppsjö av benchmark tester. I ref [12] går det att läsa om benchmark-tester. Under kategorin Open Source benchmarks finns ett 30-tal olika typer av tester, utöver dessa finns det ett antal kommersiella tester. Några utvalda benchmarks är LMbench, Real-feel Ltp är alla benchmark-tester för testning av Linux kärnan.

LMbench är en uppsättning av enkla, portabla microbenchmarks för UNIX/POSIX. Det mäter i huvudsak latens för till exempel kontextbyten utöver latens finns ett antal tester för något som i det här sammanhanget kallas minnesbandbredd och vid närmare studier inser man att det att det är tester för minneshaneting [9]. LMbench räknas till gruppen syntetiska benchmarks och saknar bland annat tester för Processer och interprocess kommunikation.

Linux Test Project (LTP) är ett samarbetsprojekt som startades av SGI™ och underhålls av IBM®. Det har som mål att leverera tester till Open Source Community. Tanken med testerna är att de ska hjälpa till att validera Linux pålitlighet, robusthet och stabilitet. Sammanfattat är LTP en samling med över 2000 tester och verktyg för testning av Linuxkärnan och relaterade funktioner.

Inom LTP finns en samling tester som går under namnet LTP stress. Denna samling använder sig bland annat företaget Monta Vista för stresstestning av sin kärna [16]. LTP stress innehåller stress tester på den mest intressanta funktionaliteten i ett RTOS. Om ett urval av dessa tester anpassas till prestandamätning kan det att bli en mängd relevanta prestandatester. Det som saknas i LTP stress (förutom tidmätning) är tester för schemaläggaren och avbrottshantering [15].

Eftersom ingen av ovanstående benchmark testsviter är möjliga att på ett rättvisande sätt köra på både OSE och Linux behövs ett antal gemensamma tester för att avgöra de två operativsystemens prestanda.

Helst bör en uppsättning benchmark tester resultera i ett oberoende mått på prestanda, vilket helst ska kunna ge möjlighet till jämförelse mellan olika typer av system. Benchmarking har i datorsammanhang ofta fått en dålig klang. Detta beror på att datortillverkar har en lång historia av att försöka sätta upp sina system på ett sådant sätt att benchmark tester resulterar i orealistiskt positiva resultat. Målet med mitt arbete är att tydligt visa att de olika testerna motsvara varandra i största möjliga mån och därmed få ett antal tester som jämför äpplen med äpplen och inte med apelsiner.

3 En testsvit för realtidsoperativsystem

Ett operativsystem tillhandahåller en miljö för exekvering av program. Operativsystemet erbjuder ett antal tjänster till andra program och/eller till användaren av dessa program. De tjänster som erbjuds varierar självklart från system till system. Men det finns ett antal klasser som är lika oberoende av typen på systemet. För ett RTOS eller ett generellt OS är grundbehovet det samma. Det som skiljer kan vara vad som är viktigt för de olika systemen. Följande punkter har Silberschatz et.al i ref [2] listat som nyckelpunkter i alla OS.

- Programexekvering: Systemet måste ha möjlighet att ladda och köra program. Program måste ha möjlighet att avslutas, antingen normalt eller onormalt.
- Input Output (I/O) operationer: Ett program kan behöva I/O. Det kan innebära aningen via en fil eller någon I/O enhet.
- Minnesmanipulation: Program måste under körning kunna läsa och skriva till minne.
- Kommunikation: I många situationer behöver en process kommunicera med en annan. Kommunikationen kan delas in i två huvudgrupper. Kommunikation som sker inom samma dator eller kommunikation som sker mellan olika datorer. Kommunikation kan implementeras till exempel via meddelanden.
- Felhantering: Systemet måste kontinuerligt vara beredd på möjliga fel. Felen ska kunna hanteras oavsett om det uppstår i CPU, minne, I/O enheter, avsaknad av nätverk eller i samband med användning av ett program.

Utöver dessa punkter är de flesta system även i behov av resursallokering, säkerhet och olika användare till systemet. För att kommunicera med systemet behövs ett gränssnitt mellan processer och OS. Gränssnittet är systemanrop. Det finns ett antal typer av anrop som behövs för att täcka in den grundläggande funktionaliteten, listad ovan.

- Processkontroll:
  - avsluta, avbryt
  - ladda, exekvera
  - skapa, döda
  - vänta
  - allokera och deallokera minne

- Filhantering
  - skapa och radera filer
  - öppna, stänga
  -läsa, skriva
  -sätta och ta bort attribut

- Enhetshantering
  - begära tillgång till enhet, släppa enhet
  - läsa och skriva till enheten
  - attacha enhet deattacha enhet

- Informationshantering
  - få tid/datum, sätta tid/datum
  - få systeminformation, sätta systeminformation

- Kommunikation
  - skapa eller avlägsna kommunikationskopplingar
  - skicka eller ta emot meddelande

Generella bitar för ett OS är alltså bland annat processhantering, schemaläggning, kommunikation och I/O. Till de viktigaste systemanropen hör processkontroll som skapar och dödar processer till detta behövs allokering och deallokering av minne. I samband med hantering av processer behövs möjligheter till att hantera den information som genereras till detta behövs funktionalitet så som minneskopiering. För att möjliggöra effektiv processhantering behövs även möjligheter till kommunikation mellan processer [2].

En typisk realtidsapplikation är den mjukvara som finns i en mobiltelefon. En mycket förenklad mobiltelefon kan tänkas bestå av ett huvudprogram och tre processer. En process som registrerar knapptryckningar från användaren. En andra process tar emot inkommande samtal och skickar en signal till högtalaren. Den tredje processen är högtalaren som väntar på en signal och när denna kommer spelar upp ett ljud. I den här situationen behövs en loop som efter uppstart väntar på att någon av processerna ska bli aktiva. När ett samtal kommer till telefonen väcks den processen och en signal sänds till högtalaren vilken börjar spela upp ljud. Samtidigt som telefonen ringer måste användaren kunna trycka på knappar för att svara eller avvisa samtalet.

För att genomföra den här enkla applikationen behövs ett antal funktioner. Det behövs, processer, dessa måste kunna skapas och avslutas. I samband med att processerna skapas behövs minne allokeras och sedan släppas. Det behövs möjligheter att skicka meddelande mellan processer. Utöver detta är det viktigt att rätt process får tillgång till processorn. Knapptryckningar måste prioriteras innan ljuduppspelning vilket utgör ett behov av schemaläggning och prioritetsnivåer.

Kapitel 2.1 och 2.2 visade den viktigaste funktionaliteten för ett RTOS och den funktionaliteten stämmer bra överens med dessa behov. De stämmer också med de punkter som Silberschatz tagit upp som de viktigaste i alla OS. Sammanfattande är alltså de viktigaste egenskaperna i ett RTOS, förutom den grundläggande funktionalitet som är lika för alla OS, är förutbestämbarhet och hastighet det viktigaste.

Det syns tydligt att det är viss funktionalitet som återkommer som den viktigaste både för RTOS generella OS och i exemplet ovan. Jag har därför valt att testa ett antal av dessa viktiga OS primitiver. Det gör jag genom att mäta tidsåtgången för några vanliga systemanrop och schemaläggning.

3.1 Operativsystemsprimitiver

Operativsystemsprimitver är den uppsättning systemanrop som täcker det primära grundbehovet av funktionalitet i ett operativsystem. Utan dessa är det omöjligt för operativsystemet att upprätthålla ens grundläggande funktionalitet. Till denna kategori av primitiver hör allokering och deallokering av minne. Utöver dessa finns även primitiver för kopiering och flytt av data. Följande primitiver är särskilt viktiga.

- *malloc(), allokerar* minne.
- *free(), återställer* allokerat minne.
- *memcpy(),* kopierar data från en buffert till en annan, ej överlappande.
- *memmove()* flyttar data från en buffert till en annan, överlappning tillåten.
- *memset(), fyller* det angivna området med angivet tecken.
- *calloc(), allokerar* minne och fyller det med angivet tecken.

3.1.1 Tester för operativsystemsprimitiver

Testerna för operativsystemsprimitiver innefattar test av *malloc(), free(), memcpy(), memmove()* och *memset(),* men inte *calloc(). Calloc()* testas inte eftersom det består av två anrop, *malloc()* och *memset(),* som båda testars var för sig.

Eftersom *malloc(), free(), memcpy(), memmove()* och *memset(),* finns i båda operativsystemen kunde samma testkod användas. Det enda som skiljer är funktionaliteten för tidmätningen. Grundstrukturen för testerna av *malloc(), free(), memcpy(), memmove()* och *memset()* ser ut enligt följande. En loop upprepar systemanropet ett stort antal gånger (mellan 500 och 10000 gånger, antalet konfigureras i testmiljön). Den stora mängden iterationer är till för att minska testresultatens känslighet för störningar. I varje loop sker två tidmätningar, en före systemanropet, och en efter.

Malloc()-testet gör tidmätningar på *malloc()* anropet med storlekar genererade av en kvasirandom funktion tillsammans med seed talet 123. Antal test iterationer och storleks intervall anges i funktionsanropet till testet. I bland testresultaten finns två tester *malloc()*-small och *malloc()*-large. I testet *malloc()*-small varierar storlekarna mellan 1 och 16*1224 bytes, vilket gör att minne kan allokeras från heapen. I *malloc()*-large varierar storlekar mellan 16* 1224 och 1 000 000 bytes, vilket gör att minne behöver allokeras från primärminnet (MM).

Test *free()* testar tiden för anropet *free()* på storlekar mellan 1 och 1 000 000 bytes. Storlekarna är genererade av samma kvasirandom funktion som *malloc()*-testerna och med seed talet 123. Testet utförs ett stort antal gånger. Antal testiterationer och storleksintervallet för hur mycket som skall allokeras bestäms utifrån testanropet. För att kunna återupprepa testet en stor mängd gånger utan att det tillgängliga minnet sätter begränsningar, så allokeras minne i omgångar. Det sker på följande sätt, först allokeras minne upp till en angiven storlek som sedan följs av tidmätningar av *free()*-anropet, detta återupprepas tills antalet testiterationer har uppnåtts. Genom att först allokera en mängd minnesområden för att sedan mäta tiden för *free()*-anropet gör att operativsystemet belastas på ett mindre fiktivt sätt än om samma minnesområde skulle allokeras och sedan omedelbart göra tidmätningen för deallokeringen.

Anropen *memcpy() och memmove()* flyttar data inom en buffert. Skillnaden mellan *memcpy()* och *memmove()* är liten. *Memcpy()* är inte definierad för att flytta data i situationer där förflyttningen kan leda till att data överlappar sig själv. I dessa situationer så behöver *memmove()* användas.

Båda testerna för *memcpy()* och *memmove()* utför en flytt på en mängd data inom en buffert. Bufferten är dubbelt så stor som datat. Efter flytten sker en kontroll så att datat som flyttats inte blivit korrupt. *Memcpy()*-testet består av två tester, ett test som flyttar från början av bufferten till slutet, det andra testet flyttar samma mängd data från slutet till början av bufferten.
Memmove()-testet är uppbyggd på samma sätt och datat flyttas från en lägre adress till en högre och tvärt om, men till skillnad från *memcpy()*-testet sker förflyttningarna med överlappning av datat i bufferten.

Det sista testet av operativsystemsprimitiverna är testet för *memset()*.
Memset-testet använder *memset()* och fyller en buffert med talet 123 och mäter tiden för denna operation.

3.2 Processprimitiver

I arkitekturer med delat minne och flera processorer används trådar och processer för att möjliggöra parallellism. Informellt kan en process beskrivas som en instans av ett program under körning eller en samling datastrukturer som beskriver exekveringen.

Från kärnans synvinkel är processens uppgift att verka som en enhet dit systemresurser som CPU-tid och minne kan allokeras [2]. En process skapas av operativsystemet, och varje ny process ger en viss overhead. I OSE finns det bara en typ av processer men i Linux finns möjlighet att skapa flera typer av processer, kallade trådar, processer och lättviktsprocesser. OSE processer kan jämföras med lättviktsprocesser.

Från början fanns det bara en typ av processer i Unix system. När en sådan process skapas blir alla resurser som ägs av föräldern duplicerade till barnet. Det gör skapandet av nya processer onödigt långsamt och ineffektivt.

Orsaken till att det är ineffektivt, är att barn-processen oftast inte behöver läsa eller ändra i alla de ärvda resurserna. I realtidssystem passar den här typen av processer dåligt. I realtidssystem är det viktigt att kontextbyte sker snabbt och den här typen av processer blir för tunga och klumpiga. För att minska overheaden har flera typer av processer utvecklats. I Linux finns det processer, lättviktiga processer och trådar. Skillnaden mellan de olika typerna är i princip hur mycket av informationen som dupliceras eller delas. Detta är intressant eftersom mängden duplicerad information påverkar tiden det tar att skapa en ny process och tiden för varje kontextbyte med den processen [13, 14].

3.2.1 Processprimitiver i OSE

Det finns bara ett rekommenderat anrop för att skapa nya processer i OSE det är *create_process()*. Det motsvarar i Linux, en lättviktig process. c*reate_process()* är lättviktig eftersom den endast skapar en ny stack och en PCB till den nya barnprocessen, resten är delat med föräldraprocessen. Utöver detta så skapar *create_process()* en stoppad barnprocess vilket leder till att föräldrarprocessen fortsätter att köra utan ett kontextbyte efter *create_process()* anropet. Vilket kanske inte alltid är av vikt men just i situationen då själva anropet ska testas är det bra att slippa ett onödigt kontextbyte, eftersom ett kontexbyte skulle ha påverkat tidmätningen.

3.2.2 Processprimitiver i Linux

När en process skapas med hjälp av *fork()* blir den en nästintill identisk med sin förälder. Barn processen får en logisk kopia av förälderns adressrymd och exekverar samma kod, med start på följande instruktion efter systemanropet som skapande barn-processen. Trots att barnet och föräldern delar samma programkod så har de separata kopior av stack och heap. Detta leder till att förändringar i minnesrymden gjorda av barnet inte påverkar föräldern och tvärt om. Detta kan ju vara bra i vissa fall men oftast leder det till onödig overhead.

I lättviktsprocesser delar barn och förälder sidtabeller och adressrymd. De delar också tabeller med öppna filer och signaler. Lättviktsprocesser skapas med hjälp av *clone()*. Service funktionen *sys_clone()* implementerar själva *clone()*-anropet. *Clone()* är endast en wraper, ett namn, som definieras i C biblioteket och som vid anrop sätter upp en stack för en ny lättviktsprocess och som tar *fn och arg* som inparametrar. I den nya stacken sparas *fn* och *arg*-pekarna och när *clone()*-anropet är klart fångas *fn(arg)* och exekveringen fortsätter från den raden. Det som gör *clone()* mest likt OSEs *create_process()* i det här sammanhanget är att det är en lättviktsprocess och att med hjälp av de argument funktionen tar är det möjligt att påverka hur tung processen och därmed processbytet blir. Utöver detta är det möjligt att skapa nya stoppade processer vilket gör att i själva testet undviks ett onödigt kontextbyte.

Det traditionella systemanropet *fork()* är i Linux implementerat på samma sätt som *clone()*. Det som skiljer *fork()* från *clone()* är att *fork()*-anropet inte tar parametrar som specificerar SIGCHILD signaler eller andra flaggor. Vidare skiljer sig *fork()* från *clone()* eftersom barnets stackpekare blir den samma som den nuvarande stackpekaren. Detta medför att barnet och föräldern temporärt delar samma User Mode Stack. Det är först när någon av dem försöker ändra i stacken som en ”Copy On Write mechanism” skapar två separata skrivskyddade kopior av stacken. Det är först när någon försöker skriva till stacken som själva kopieringen sker.

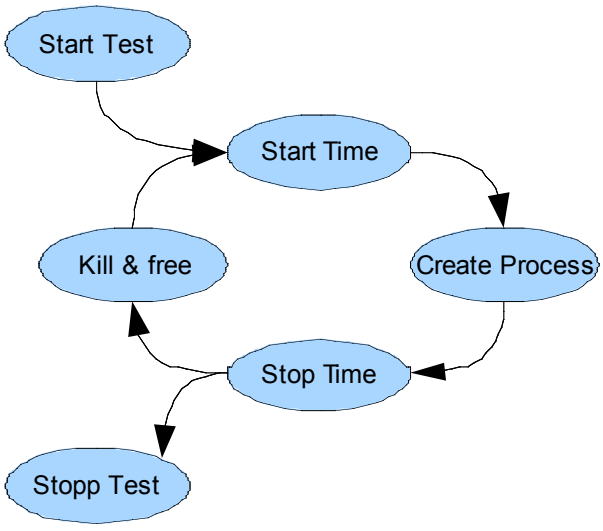
Den tredje typen av systemanrop för skapandet av processer är *Vfork()*. *Vfork()* är implementerad som en *clone()* med flaggor som specificerar SIGCHILD-signaler, men barnets stackpekare är densamma som föräldern. Med systemanropet *Vfork()* skapas en process som delar hela minnets adressrymd med föräldern. För att skydda data från sönderskrivning blir föräldraprocessen stoppad tills barnet har exekverat klart [3, 13].

Pthreads refererar till en POSIX standard vilken definierar ett API för tråd skapande och synkronisering. Det är endast en specifikation för trådbeteende, inte en implementation [2].

Den av de tre typerna som kan göras mest lik OSEs *create_process()* är *clone()*-anropet. De största likheterna ligger i att med hjälp av flaggor kan stackstorleken definieras och även att processen kan skapas i ett läge där den förblir stoppad. Vilket leder till att kontextbytet kan kontrolleras. I OSE sker inte heller något kontextbyte eftersom barnprocessen skapas stoppad.

3.2.3 Tester för processprimitiver

De viktigaste aspekterna av att bedöma prestanda i ett operativsystem är att avgöra vad som är intressant för varje enskild situation. Vad som sker ofta i ett system kan vara väldigt ovanligt i nästa. Skapa processer är en operation som sker ofta likaså operationen som avslutar processer. Det går att resonera som Gallmeister att *”The speed with which you can create and terminate processes is not usually time-critical, since it's not something you generally do in the middle of real-time operations”* ref [1]. Det är några år sedan Gallmeister skrev detta och nu mera är det vanligt att processer skapas under körning även i ett realtidssystem. Därför är det finner jag det intressant att mäta både skapande och terminering av processer. Vidare är jämförelsen intressant eftersom det blir omöjligt att göra en bedömning av schemaläggaren utan kunskapen om hur mycket de olika typerna av processer skiljer sig åt.
I testet för *create* används i OSE *create_process()* och jämförs med *clone()* i Linux. Storleken på minnesbufferten som kopieras är den samma i båda anropen. I *clone()* finns det en mängd växlar för inställningar av vad som kopieras. I testet har jag använt alla växlar som gör *clone()* mer lättviktig och på så sätt gjort *clone()* så lättviktig som möjlig. Utöver detta skapas barnprocessen i ett stoppat läge för att undvika ett kontextbyte, precis så som i OSE. I testet för *kill()* mäts tiden det tar att döda motsvarande process, alltså *kill()* på en *clone()* i Linux.



Figur 2: Testet för Create Process

3.3 Inter-processkommunikation

Processer som kan exekveras samtidigt i ett system är antingen oberoende eller samarbetande processer. En process räknas som oberoende om den inte kan påverkas eller påverka andra exekverande processer i systemet. Det innebär alltså om en process inte delar någon data temporärt eller persistent med någon annan så räknas denna process som oberoende. En samarbetande process kan bli påverkad och påverka andra exekverande processer i systemet. Samtidigt exekverande processer som samverkar kräver mekanismer som tillåter kommunikation och synkronisering mellan processerna.

Det finns två huvudtyper av kommunikationsmöjligheter för samarbetande processer. Kommunikationen kan ske med eller utan synkronisering via operativsystemet.
Sker kommunikationen med hjälp av delat minne har processerna en gemensam minnespool. Synkronisering och säkerhet appliceras endast av den applikation som styr processerna.

Alternativt kan kommunikationen ske med hjälp av verktyg som operativsystemet bistår med. Det blir då operativsystemets uppgift att lösa synkronisering och eventuell säkerhet för dessa mekanismer. Silberschatz, et.al definierar i ref [2] inter-processkommunikation (IPC) som ”*IPC provides a mechanism to allow processes to communicate and synchronize their actions without sharing the same address space.*” IPC är alltså som den grupp av mekanismer som möjliggör synkronisering och kommunikation mellan processer utan att de behöver dela samma adressrymd.

IPC i OSE sker med hjälp av OSE signaler. I Linux finns flera typer mekanismer för kommunikation implementerade. För att finna den typ som är mest lik OSE signalen har några olika typer undersökts.

3.3.1 Inter-processkommunikation i OSE

OSE signaler kan skickas till alla processer med ett känt processid. Storleken på meddelanden är endast begränsad av poolstorleken. När poolen är fylld med olästa meddelanden kan inte fler skickas innan något har blivit läst av sin mottagare. Det går inte att sätta prioritet på meddelande vilket det går i POSIX 4 meddelanden, men annars anser jag att de är så lika att den blir en intressant jämförelse mellan de två [14].

3.3.2 Inter-processkommunikation i Linux

Direkt kommunikation i Linux kan ske med till exempel signaler. Dessa kommer i fortsättningen kallas för Linux signaler för att hålla de åtskilda från OSE signaler. Med Linux signaler fungerar kommunikationen med hjälp av heltalsmeddelanden. Ett meddelande består av ett heltal mellan 0-99. Till varje heltal finns ett namn koppalt. Alla heltal, utom två, är kopplade till ett vist systemanrop. De som kan definieras av applikationen är SIGUSR1 och SIGUSR2. Den här typen av kommunikation används till asynkron information till processer och avbrottshantering.

Direkt kommunikation med hjälp av Linux signaler skiljer sig från OSE signaler på flera punkter då dessa bland annat har en mycket begränsad meddelandestorlek, samt att de är mer anpassade för avbrottshantering än meddelandekommunikation. Det blir alltså ingen bra jämförelse att mäta prestanda på dessa mot OSEs signaler.

Med indirekt kommunikation kan meddelanden skickas och mottas via brevlådor. Rent generellt är meddelandeköer mer lika OSEs signaler än Linux signaler. Det finns några olika implementationer av meddelandeköer. Exempel på dessa är pipe, fifo, system V- och posix 4 köer.

Pipe består av två fil deskriptorer en för skrivande och en för läsande. Det finns ett begränsat antal fil deskriptorer och varje kan bara antingen användas till läsning eller skrivande. Kommunikationen sker enligt följande princip. I den läsande änden kan den mottagande processen läsa en byte i taget. I den skrivande änden fyller processen på en byte i taget tills meddelandet är slut eller pipen är full. Det går att fylla på flera meddelande tills pipen är full. Eftersom den läsande processen endast läser ett tecken åt gången finns inga garantier att den läsande processen har sett hela meddelandet.

Fifo är en namngiven pipe. All funktionalitet i fifo är densamma som i en pipe, förutom möjligheten att referera till den med dess namn. Namnet på en namngiven pipe är endast ett filnamn definierat av operativsystemet. Det ger en effekt av att det blir möjligt att köra samma typer av kommandon på kön som på andra filer, det vill säga till exempel *ls* för att lista innehållet.

Pipe och FIFO är mer avancerade än Linux signaler men den begränsade meddelande säkerheten och att antalet fil deskriptorer är begränsat samt att det behövs två för tvåvägs kommunikation gör dessa till dåliga kandidater för ett intressant test.

System V:s meddelandeköer är något mer avancerade än pipe och FIFO. Med system V köer finns det möjlighet till tvåvägs kommunikation. En system V meddelandekö identifieras med ett unik positivt heltal. Processer kan accessa resursen genom att skicka denna unika referens till kärnan via ett systemanrop. Efter anropet är det sedan möjligt att accessa kön. Varje gång en process försöker skriva ett meddelande kontrolleras processens gruppmedlemskap samt skriv och läs rättigheter till kön. Avsändande och mottagande process behöver inte existera samtidigt för att nyttja kön, vilket de måste gör i en pipe/FIFO kommunikationssituation. Liksom pipe/fifo finns storleks begränsningar. Begränsningarna definieras när kön sätts upp. Det finns begränsningar för varje enskilt meddelande och på kön som helhet. Den maximala storleken varierar beroende på version men för meddelande verka maximalt 1kb och för hela kön 16kb vara det vanligaste [3, 11].

I ref [1] beskriver Gallmeister system V meddelandeköer som en ”*evolutionary oddity* ” och beskriver dem även som tunga och otympliga men att de till trots detta ger den viktigaste funktionalitet som behövs för IPC i ett realtidssystem. Vidare talar han för POSIX 4s meddelandeköer och beskriver dessa som bättre eftersom de är utformade för att användas för IPC inom realtids applikationer. System V meddelande köer skulle kunna jämföras med OSEs signaler då de har den mesta funktionaliteten som signaler. Men eftersom Gallmister beskriver dessa som tunga samt POSIX 4 signaler som anpassade för realtid applikationer kommer dessa att används.

POSIX 4 meddelandeköer finns i Linux kärnan från version 2.6.6. De är uppbyggda som andra meddelandeköer men ger en möjlighet att definiera fler parametrar vid start. Skillnader är till exempel meddelande storlek över 1kb och prioritet på meddelande. Det går även att använda dessa meddelandeköer i ett distribuerat system då det inte spelar någon roll var processerna befinner sig. Meddelandekön existerar inte i filsystemets namnrymd vilket medför att kön kan fortsätta existera även efter de processer som använder den har terminerat. Det finns funktionalitet för att sätta upp kön så att mottagaren blir informerad när ett meddelande väntar. Storlek på själva kön är en bieffekt av att vid start bestäms den maximala storleken på varje meddelande samt att antalet olästa väntande meddelande. Referensen till kön är den samma som dess absoluta sökväg i filsystemet [1, 3].

3.3.3 Test av POSIX 4 meddelandeköer mot OSE signaler

Gallmeister skriver i ref [1] att effektivitet och prestanda hos signaler är mycket viktig för ett RTOS, om systemet använder signaler. Det intressanta att mäta är i så fall hastigheten för signalen att nå sitt mål. Vidare skriver Gallmister att POSIX 4 meddelande köer är avsedda för realtids systems meddelande hantering. För att testa Linux signaler förslår Gallmeister tester på receive och på send. Efter samtal med utvecklare på OSE föreslår jag tre tester ett send test och ett send-receive-send-receive test samt ett receive test.

Test *send()* är ett mycket enkelt test avsett att mäta själva *send()* funktionen med så lite overhead som möjligt.
En tidmätare placeras innan *send()*-anropet och en efter . I Linux måste meddelandestorleken definieras så den motsvarar OSEs meddelandestorlek. Tiden mäts för 1000 meddelande med storlekarna 1, 5 och 16kb.

Testet för send-receive-send-receive är mer avancerat än det föregående men mera ”verklighets” anpassat. Det går ut på att mäta en hel loop av *send()* och *receive()*. Här kommer det att bli en del overhead men genom att repetera testet ett stort antal gånger kommer ändå resultatet vara intressant.
Skapa två processer och i Linux en meddelandekö. Sätt tidmätaren innan första *send()*. Sätt den andra processen att bevaka kön och låt den göra en reply när meddelandet når denna process. När den första processen får svaret stoppas tidmätningen. Tiden mäts för 1000 loopar med meddelande i olika storlekar.

3.4 Schemaläggning

Schemaläggning är särskilt viktig i ett realtidssystem. Resurser behöver schemaläggas innan användning. CPUn är en primär resurs och dess schemaläggning blir därför central för hela operativsystemets prestanda. När processorn blir overksam tas en körbar process ur redo-kön. Detta sköts av schemaläggaren. Det finns olika typer av schemaläggningsalgoritmer anpassade för olika typer av system. I Hårda Realtidssystem[8] använder schemaläggaren sig av resursreservation. Men eftersom varken OSE eller Linux använder denna typ av algoritm kommer den inte vidare att nämnas här. I mjuka realtidssystem[9] finns två intressanta faktorer som påverkar schemaläggarens effektivitet, det är möjligheten till process prioritet och dispatch latens. Silberschatz, et al skriver i ref [2] att implementering av schemaläggaren i ett mjukt realtidssystem kräver noggrann design, där processer behöver ha prioritet och realtidsprocesser ska ha den högsta prioriteten. Utöver detta måste dispatch latensen vara låg. Kortare dispatch latens[10] gör att en realtidsprocess kan börja exekvera snabbare. Det är relativt enkelt att garantera att prioritetsordningen följs. Men att begränsa tiden för dispatchen mer komplicerad. Problemet är att I/O enheter är långsamma och att även komplicerade systemanrop kan lång tid. Detta leder till att processbytet kan dra ut på tiden. Det går att begränsa tiden för en dispatch med hjälp av till exempel premtion points. Men det är inte intressant här då det endast är slutresultatet av tiden för själv kontextbytet som ska mätas.

3.4.1 Schemaläggaren i Linux

Schemaläggaren i Linux erbjuder tre typer av schemaläggnings regler en för normala processer och två för realtidsapplikationer. Ett statiskt prioritetsvärde ges till varje process vid uppstart. Prioritetsvärdet kan endast ändras via ett systemanrop. Konceptuellt håller schemaläggaren en lista över körbara processer för varje prioritetsnivå. Prioritetsvärdena kan variera mellan 0 och 99. För att bestämma vilken process som ska få tillgång till processorn näst letar schemaläggaren efter den första icke tomma listan med högsta prioritet. Schemaläggaren bestämmer för varje process, var den ska placeras i listan av processer med samma prioritet, och hur processen ska förflyttas i listan [3].

3.4.2 Schemaläggaren i OSE

Schemaläggaren i OSE har två typer av schemaläggnings regler. En för prioriterade processer och en för icke prioriterade processer. De prioriterade processerna schemaläggs enligt strikt prioritetsordning, och avbryts inte. De icke prioriterade processerna schemaläggs enligt Round Robin.

---

8 Hård realtid innebär att en missad deadline resulterar i förödande konsekvenser för systemet. Alla tidsbegränsningar måste hållas oavsätt situtation

9 Mjuk Realtid innebär att en missad deadline inte resulterar i förödadnde konsekvenser.

10 Dispatch latens är tiden mellan det att en process blir schedulerad för att få köra tills dess att den börjar exikveras.

3.4.3 Test av schemaläggaren

I alla operativsystem med fler än en process är schemaläggning av CPUn en fundamental funktion för systemets prestanda. Schemaläggaren avgör vilken process som ska köras när processorn blir ledig [2]. Gallmeister beskriver några tester av schemaläggning bland annat ett yield test som är intressant i det här sammanhanget [1].

Test av *yield()*-anrop. Med *yield()* lämnas en process frivilligt över processorn till nästa i redo-kön. Detta ger ett frivilligt kontextbyte. Det innebär att inget interrupt är inblandat och tar extra tid. I Linux finns det ett explicit kommando för *yield()* . I OSE ger funktionen *set_prio()* på den aktuella processen samma effekt. Detta förutsätter att prioriteten sätts till samma värde som processen redan har. Finns det bara en process kommer den att få tillgång till processorn igen vilket gör att det går att mäta tiden för kontextbytet genom att starta en timer innan *yield()*-anropet sedan på nytt mäta tiden när processen blir aktiv igen.

Testet för bakgrundsprocesser är utformat på följande sätt. Genom att först skapa 50 processer med i Linux låg prioritet, eller i OSE, bakgrundsprocesser utan prioritet. Sedan sätts alla processer att utföra en meningslös loop. I den processen första loop görs en tidmätning och lika så i den 50e. Tidsskillnaden blir då kontextbytet plus timeslotten för varje process.

3.5 Testmiljö

- Testerna är utförda på ett mercury MPQ101kort med en freescale mpc8548erm processor. Kortet sitter i en Ensemble Advanced TCA Application Platform och benämns i linx labbet som pq3.
- LINUX version 2.6.13-MPQ-101 (chayden@ccg1) (gcc version 3.4.4) #4 Wed Aug 9 14:29:51 EDT 6
- OSE 5.3 (BL140069)

4 Resultat

Här följer en sammanfattande analys av testresultaten, diagram och testresultat finns inte i denna nedkortade rapport utan fås mot förfrågan.

4.1 Memcpy

Den tidsskillnad som finns mellan OSE och Linux härrör till att i Linux är *memcpy()* implementerad med hårdvaruspecifik assembler medan i OSE är *memcpy()* implementerade med generisk C-kod.

Fördelen med en assembler implementation är att det ger större möjligt för tidsoptimeringar. Nackdelen med en optimerad assembler lösning är att den kräver en unik implementation för varje hårdvara. Den generiska C kods lösningens nackdel är dock att den inte är lika tidsoptimerad.

4.2 Memmove

Memmove() är precis som *memcpy()* implementerad med assembler i Linux men C i OSE. Det som skiljer det här testresultaten från *memmove()* ovan är att Linuxkärnans assembler implementation inte är optimerad för hårdvaran. En trolig anledning till att *memmove()* inte är tidsoptimerad i Linuxkärnan är att den endast används en gång i boot up sekvensen.

En ooptimerad hårdvaruspecifik assemblerlösning kan alltså ta längre tid än en generisk C-implementation.

4.3 Memset

I Linux är memset implementerad med assembler och i OSE i generisk C- kod. Linux implementationen är bättre optimerad än i *memmove()* vilket syns i testresultaten. Utöver att tidsoptimerad assembler blir snabbare än C-kod går det att tillägga att både OSE och Linux är tämligen deterministiska. Vilket är en viktig egenskap i realtidssammanhang.

4.4 Malloc large och Malloc small

Testet m*alloc()*-large är utformat så att minne behöver hämtas från MM, och testet *malloc()*-small” hämtas minne från heapen. I OSE finns ett konfigurerbart tröskelvärde för när minne ska hämtas från MM, vilket ger möjlighet att optimera operativsystemet för olika behov. I de situationer när minne inte hämtas från MM är OSE både snabbt och deterministiskt men när MM aktiveras minskar determinismen och tiden ökar betydligt. I Linux är determinismen begränsad men det värsta fallet på de två testerna har endast en försumbar tidsskillnad.

4.5 Free

Det finn två typer av *free()* algoritmer den ena typen är optimerad för tid den andra för storleks effektivitet. I OSE används en storleksoptimerad algoritm. Men trots detta klara sig OSE förvånansvärt bra mot Linux i det här testet, där Linux är på tok för odeterministisk.

4.6 Clone/Create process

Begreppen processer och trådar skiljer sig mycket åt i Linux och OSE. Men som jag tidigare beskrivit är *clone()* och *create_process()* likvärdiga. Skillnaden mellan de två är att i *clone()* sker en kopiering av sidtabeller. Den extra kopieringen av sidtabellerna i *clone()* borde göra att det tar längre tid än *create_process()*. Trots kopieringen är dock Linux något snabbare. Båda operativsystemen är mycket deterministiska med i Linux varierar resultaten något mer än OSEs resultat. I resultatet från OSE-testet finns endast en förekomst av 47 mikro sekunder, annars helt stabilt på 11u sekunder. Eftersom det är endast ett fall av en mycket avvikande tidmätning är det möjligt att det förekommit någon form av störning i systemet. Risken med den här typen av tester är att det kommer okontrollerbara störningar, men alternativet skulle vara att göra tester i ett helt rent system, men det skulle leda till mer verklighets främmande resultat. I verkligheten förekommer störningar och ett operativsystem för känsliga miljöer måste klara av även dessa störningar.

Förutom den misstänkta störningen är resultatet för OSE något långsammare, men mer deterministiskt än för Linux.

4.7 Kill

Båda operativsystemen visar ett deterministiskt beteende men för OSE tar det mycket lång tid att döda en process.
En process i OSE som körs i user- mode utan att någon prioritet är satt, får en lägre prioritet än systemdemonen. Det medför att systemdemonen utför anropet först när den körs. Om processen hade haft en högre prioritet än systemdemonen hade anropet utförts enligt korrekt prioritets ordning. Detta hade antagligen lett till en betydligt kortare tid för OSE. I Linux släpps en pekare och minnet frigörs först när det behövs för någon annan allokering.

4.8 Send och Receive

Det testresultaten visar är att i Linux påverkas tiden för Send-Receive av mängden data i meddelandet medan i OSE är tidsåtgången konstant. Detta är en sanning med ett visst förbehåll.

När data skickas mellan processer inom samma segment i OSE görs ingen kopiering medan detta sker i Linux. För att få testerna jämbördiga borde därför OSE testet ha skickat data mellan två segment. Att simulera två segment visade sig vara svårare att göra än vad det föst verkade. Tillslut tvingades jag ge upp försöken att simulera två segment eftersom det inte fanns tid inom ramen för detta arbete.

För att ge en mer rättvisande blid över tidsåtgången har jag med hjälp av testresultaten från *memcpy()* gjort en beräkning över ett simulerat resultat. Här syns det tydligt att det tar mer tid att skicka ett större meddelande än ett litet. De beräknade resultaten visar att är konkurrens kraftigt med Linux och att de framför alt är mycket mer deterministiska. I det här testet visar Linux på en oförsvarbart låg determinism.

4.9 Yield

OSE är optimerat för determinism vilket har lett till en sämre genomsnitts tid. Linux varierar kraftigt med nära en fördubbling av tiden mellan det snabbaste testet och det långsammaste.

4.10 Scheduler

Testet för schemaläggaren bygger på att mäta tidsåtgången för 50 processer har under utvärderingen visat sig inte vara så genomtänkt det verkade vid design och implementationen. Problemet med testet är att det är svårt att avgöra i testsituationen hur mycket tid som processorn varit overksam för att vänta på att sovande processer.

Om tidmätningen skett när till exempel hälften av processerna var klara hade det här testet varit mycket mer tillförlitligt.

5 Slutsatser

Testerna för operativsystems primitiver, eller det vill säga testerna för *malloc(), free(), memcpy(), memmove()* och *memset(),* men inte *calloc(). Calloc()* testas inte eftersom det består av två anrop, *malloc()* och *memset(),* och båda dessa testas var för sig. Resultaten visade att båda operativsystemen har hög determinism. Ur de uppmätta tiderna ser man att de två operativsystemen är bra på olika saker.

I testerna för process primitiver testades tiden för att skapa och avsluta processer detta gjordes i testerna *create_process()*/*clone()* och *kill()*. Resultaten av dessa tester visar på en förvånande icke determinism.

Interprocesskommunikation är lite mer osäker att uttala sig om eftersom resultaten baseras på ett kalkylerat värde för OSE, men detta till trots är det en slående stor tidsskillnad mellan de två operativsystemen.

I gruppen av tester för schemaläggning kommenterar jag bara testet för Yield. Åter igen syns här att de två operativsystemen är bra på olika områden. I vissa system må det vara viktigast med hastighet i andra är determinismen det viktigaste.

6 Litteraturförteckning

[1] Bill O Gallmeister, Programming for the real world POSIX.4, O'Reilly & Associates, Inc 1995 ISBN 1-56592-074-0

[2] Abraham Silberschatz, Peter Baer, Greg Gange, Operating System Concepts Sixth Edition, John Wiley &Sons Inc, 2003, ISBN 0.471-25060-0

[3] Daniel P. Bovert, Marco Cesati, Understanding the Linux kernle third edition, O'Reilly Media Inc, 2006 ISBN-10 0-596-00565-2, ISBN-13 978-0-596-00565-8

[4] Jian Yang, Yu Chen, Huayong Wang, Bibo Wang. A Linux Kernel with fixed Interrupt latency for Embedded Real-Time Systems. Proceedings of the second International Conference on Embedded Softwere smd systems ICESS, IEEE

[5] Wiliam von Hagen Real-Time and Performance Improvements in the 2.6 Linux Kernel 2005 Linux Journal, ACM press.

[6] Steve Brosky Shielded CPUs Real-Time Preformance in Standard Linux, ACM Press.

[7] Kevin Dankwart. Real Time and Linux part1 Linux Devices 2002. http://linuxdevices.com/articles/AT5997007602 ,visited 2008-07

[8] RTOS Evaluation Project, Dedicated Systems Experts What makes a good RTOS Doc no DSE-RTOS-EVA-001 Issue 2.0 2001 http://www.omimo.be/encyc/buyersguide/rtos/evaluations/docspreview.asp ,visited 2008-07

[9] LMbench - Tools for Performance Analysis http://www.bitmover.com/lmbench/ ,visited 2008-07

[10] André D. Balsa, Linux Benchmarking HOWTO 1997 http://oss.sgi.com/LDP/HOWTO/Benchmarking-HOWTO.html ,visited 2008-07

[11] David A Rusling The Linux document project. http:/tldp.org/LDP/tlk/tlk-toc.html , visited 2008-07

[12] Wikipedia http://en.wikipedia.org/wiki/Benchmark_%28computing%29 ,visited 2008-07

[13] The Linux man-pages project http://www.kernel.org/doc/man-pages/index.html ,visited 2008-07

[14] User dockumentation OSE 5.3

[15] Linux Test Project http://ltp.sourceforge.net/ ,visited 2008-10

[16] Monta Vista http://www.mvista.com/services_custom_test.php ,visited 2008-10