Timervirtualisering för hypervisors

JONAS HAGLUND

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2012

Timervirtualisering för hypervisors

J O N A S H A G L U N D

DD143X, Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik 300 högskolepoäng
Kungliga Tekniska Högskolan år 2012
Handledare på CSC var Mads Dam
Examinator var Mårten Björkman

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/
haglund_jonas_K12031.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Innehållsförteckning

1 Inledning

Ett problem i dagens samhälle är energikonsumtionen. Internet växer kontinuerligt och är därför ett växande bidrag till denna energikonsumtion. En del av Internet är dess serverhallar och över 1 % av Jordens elektricitetkonsumtion går åt till att driva dessa[1]. Samtidigt är många av dessa servrar kraftigt underutnyttjade vilket leder till onödigt energislöseri[2].

Denna ineffektivitet har dock minskat de senaste åren med hjälp av systemvirtualisering[1]. Systemvirtualisering möjliggör exekvering av flera operativsystem på samma dator. I dessa sammanhang kallas ett sådant operativsystem med tillhörande applikationer för gästsystem eller virtuell maskin, VM. Genom att exekvera flera operativsystem på samma dator minskar behovet av antalet datorer, och därmed energikonsumtionen.

Om datorns hårdvara inte är anpassad för systemvirtualisering kan det inte garanteras att gästsystemen exekverar på ett säkert sätt. Denna osäkerhet uppstår på grund av att gästsystemen skulle kunna påverka varandra under deras exekvering. För att möjliggöra säker exekvering av gästsystemen används en hypervisor. En hypervisor är ett lager mjukvara installerad mellan datorns hårdvara och gästsystemen. Hypervisorns uppgift är att skydda sig själv från gästsystemen, gästsystemen från varandra, och att schemalägga gästsystemen (för att ge alla gästsystem exekveringstid).

Med tanke på vad en hypervisor erbjuder för funktionalitet kan därmed andra fördelar med systemvirtualisering ses:

- Underhållskostnader, och mängden hårdvara och fysiskt utrymme minskar.
- Vissa säkerhetsbrister i operativsystem kan motverkas genom hypervisorns isoleringsförmåga. Exempelvis finns hypervisors i vissa inbyggda system för att skydda ett realtidsoperativsystem som utför kritiska operationer vid sidan av ett generellt operativsystem.
- Applikationer gjorda för specifika operativsystem kan användas samtidigt på samma dator vilket förenklar användningen av dessa applikationer.
- Helpdesk-support: Gör det enklare för supportpersonal att hjälpa sina kunder genom att följa kundernas handlingar.
- Plattformsbyte: Hypervisors kan emulera instruktionsuppsättningar andra än den instruktionsuppsättning som hypervisorn körs på. Detta gör att operativsystem och deras applikationer kan köras på andra hårdvaruplattformar än de är gjorda för, vilket öppnar för portabilitet.

För att systemvirtualisering skall kunna erbjuda funktionerna ovan krävs det att hypervisorn kan schemalägga gästsystemen. Schemaläggningen använder en timer som signalerar hypervisorn när ett gästoperativsystem skall bytas mot ett annat. På ett liknande sätt fungerar gästoperativsystemens schemaläggning av applikationer. Detta innebär att hypervisorn och alla gästsystem skall dela på samma fysiska timer samtidigt.

Då timern är en vital del av en hypervisor, och att en hypervisor kan erbjuda viktiga funktioner, är det intressant att undersöka hur en timerkomponent kan implementeras i en hypervisor. Därför är detta projekts mål att illustrera hur detta problem löses: *Hur kan konstruktionen se ut av den timerkomponent som finns i en hypervisor?*

En specifik problemdefinition finns beskriven i kapitel 3. Anledningen är att det krävs viss bakgrundskunskap för att förstå vilka krav en timerkomponent i en hypervisor skall leva upp till.

2 Bakgrund

För att förstå hur en timerkomponent kan konstrueras i en hypervisor behövs kunskaper om både hypervisors och virtualisering.

2.1 Virtualisering

Virtualisering av ett befintligt system innebär följande. Det befintliga systemets funktionalitet utökas, och kring detta utökade system skapas ett nytt gränssnitt. Resultatet blir att det ursprungliga systemet framstår som ett annat system det egentligen inte är.

Ett exempel är partitionering av en hårddisk. Den hårddisk applikationerna på ett operativsystem ursprungligen uppfattar. Genom att operativsystemet bidrar med funktionalitet och ett nytt gränssnitt resulterar detta i att den fysiska hårddisken framstår som flera hårddiskar. Det är dessa hårddiskar applikationerna uppfattar efter att hårddisken partitionerats/virtualiserats.

Konceptet virtualisering används av operativsystem i flera avseenden:

- Processorn: Genom tidsdelning av processorns exekveringstid kan alla applikationer exekvera som om applikationerna exekverade på en egen processor.
- Minne: Genom virtuellt minne kan varje applikation exekvera som om denne hade eget minne. Detta är dock inte fallet utan minnet är uppdelat i regioner.
- I/O: Applikationer kan ha tillgång till bland annat tangentbordet, musen och Internet (nätverkskortet).

Som skall ses kan en hypervisor liknas med ett operativsystem, och gästsystemen med operativsystemets applikationer.

För att förstå timerkomponentens roll vid systemvirtualisering bör de tre delarna av systemvirtualisering förklaras: Gästsystem, hypervisor och hårdvara.

2.2 Gästsystem

Det absolut viktigaste vid systemvirtualisering är att gästsystemen beter sig på exakt samma sätt som vid exekvering på egen dator, åtminstone angående exekveringsresultat. Då tiden måste delas mellan gästsystemen kan inte exekvering som beror på tiden förväntas vara identisk med fallet då ett gästsystem exekverar på egen dator.

Ett gästsystemen består av två komponenter: Applikationer och ett operativsystem. En applikation exekverar genom två gränssnitt: Operativsystemets systemanrop och processorns instruktionsuppsättning. Ur applikationernas perspektiv kan dessa två gränssnitt sammanfogas till ett och ge en bild av en maskin. I fallet med systemvirtualisering befinner sig hypervisorn under denna maskin vilket gör att denna maskin framstår att vara en maskin den inte är. Alltså är det en virtuell maskin.

2.3 Hypervisor

En hypervisors uppgift är att virtualisera en hel dator, det vill säga implementera systemvirtualisering. Genom att hypervisorn får datorn att framstå som ett flertal får varje gästsystem uppfattningen av att exekvera på en egen dator.

För att hypervisorn skall kunna virtualisera datorn måste denne virtualisera och skydda datorns alla hårdvaruresurser: Processor, Minne och I/O. *Detta för att dels möjliggöra samtidig användning av dessa resurser, men också för att förhindra ett gästsystems möjlighet att påverka hypervisorn och andra gästsystem*.

För att garantera att ett gästsystem inte utför skadliga operationer på andra gästsystem måste varje instruktion det exekverar kontrolleras. Detta kan uppnås på bland annat följande sätt:

- Interpretation: Hypervisorn tolkar instruktion för instruktion och de instruktioner som ej kan påverka hårdvaran exekveras. Resterande instruktioner ersätts med instruktioner som resulterar i att hypervisorn får kontroll och därmed kan emulera den ursprungliga instruktionen.
- Interpretation med cache: Likvärdig med interpretation med skillnaden att block av översatta instruktioner lagras för senare återanvändning. Detta för att möjliggöra snabbare exekvering.
- Kontrollöverföring: När gästsystemen skall exekvera en instruktion som påverkar hårdvaran sker en kontrollövergång till hypervisorn. Detta gör att hypervisorn kan emulera instruktionen per automatik.
- Paravirtualisering: Ett specificerat gränssnitt mellan gästsystem och hypervisor. Gränssnittet är konstruerat för att kringgå svårigheter i systemvirtualisering. Tekniken gör det däremot nödvändigt att modifiera gästoperativsystemen en aning, men erbjuder istället betydligt bättre prestanda än interpretation.[3]

2.4 Hårdvara

För att förstå vilken hårdvara hypervisorn måste skydda och virtualisera samt vilka krav timerkomponenten skall leva upp till, beskrivs CPU, minne och I/O.

2.4.1 CPU, Central Processing Unit

CPU:n exekverar instruktioner som styr all hårdvara i datorn.

2.4.1.1 Instruktionsuppsättning, ISA, Instruction Set Architecture

En CPU har en mängd instruktioner och register och de kan i huvudsak delas upp i två kategorier:

- Privilegierade instruktioner och register: Dessa instruktioner och register påverkar hårdvarans konfiguration. Exempelvis kan dessa instruktioner och register ändra minnesadressers åtkomsträttigheter och initiera kommunikation med I/O-enheter.[3]
- Icke-privilegierade instruktioner och register: Dessa instruktioner och register används för beräkningar och för att styra programflödet. Då dessa instruktioner och register inte kan påverka hårdvaran kallas de icke-privilegierade.[3]

Exempelvis är de båda instruktions- och registerkategorierna tillgängliga för operativsystem, då operativsystemet skall hantera all hårdvara. Dock får inte en applikation ändra hårdvaran då det skulle kunna påverka andra applikationer. Därmed är det önskvärt att applikationer endast har tillgång till de icke-privilegierade instruktionerna och registerna.[3]

Därför designar processortillverkare processorer med i huvudsak två exekveringslägen: privilegierat läge och icke-privilegierat läge. Dessa två lägen kan i sin tur delas upp i flera exekveringsnivåer. Varje exekveringsnivå definierar vilka instruktioner och register som kan exekveras respektive manipuleras i just den nivån.

Om processorn befinner sig i en icke-privilegierad exekveringsnivå och en privilegierad instruktion skall exekveras sker en kontrollövergång. Vid en kontrollövergång avbryter processorn den normala exekveringen för att exekvera instruktioner lagrade på en specifik plats i minnet. Instruktionerna på denna specifika plats i minnet är till för att hantera otillåten exekvering.[3]

2.4.1.2 Avbrott

Timerkomponenten som skall konstrueras beror på en fysisk timer som genererar avbrott då dess räknare räknat ned till noll. Därför bör konceptet avbrott beskrivas:

1. Processorn exekverar.
2. En enhet i datorn vill ha processorns uppmärksamhet och genererar därför en signal till processorn, ett så kallat avbrott.
3. Processorn tar emot signalen vilket resulterar i att den normala exekveringen pausas och processorn exekverar specifik kod för behandling av avbrottet.
4. Därefter återupptas den normala exekveringen där den slutade i steg 1.

2.4.2 Minne

Minnet används av processorn för lagring av instruktioner och data. När processorn önskar göra en åtkomst till minnet hanterar MMU:n denna (Memory Management Unit). MMU:n är en enhet i processorn som utför adressöversättning vid virtuellt minne och som kontrollerar åtkomsträttigheter till minnet.[4]

2.4.3 I/O, Input/Ouput

2.4.3.1 I/O-kommunikation

Med I/O menas överföring av data (instruktioner och variabler) mellan externa hårdvaruenheter och CPU:n och minnet. Då det finns en mängd olika sorters I/O-enheter med olika gränssnitt finns ingen standard för denna kommunikation. Därför används drivrutiner. En drivrutin är ett datorprogram som är skrivet för att en applikation skall kunna kommunicera med en specifik I/O-enhet.[3]

Då I/O-enheterna tillhör datorns hårdvarukonfiguration tillhör drivrutinerna operativsystemet. För att operativsystemet skall kunna använda drivrutinerna (som kommunicerar med hårdvara) måste dessa drivrutiner exekvera i privilegierat läge. Detta gör att drivrutiner har möjlighet att ändra en dators hårdvarukonfiguration.[3]

För I/O-kommunikation finns främst två metoder:

- Separata I/O-instruktioner: Specifika instruktioner i processorns instruktionsuppsättning som används för I/O-kommunikation. Dessa instruktioner anger vilken I/O-enhet som skall styras och vilken data som skall överföras.
- Memory-Mapped I/O, MMIO: Varje I/O-enhet tilldelas en del av minnets adressutrymme. När processorn skriver data till dessa minnesplatser kommer denna data att överförs till motsvarande I/O-enhet.[5]

2.4.3.2 DMA, Direct Memory Access

DMA används för att avlasta processorn i samband med överföring av data mellan I/O-enheter och minnet. DMA-enheten utför denna överföring genom att bli instruerad av processorn om vilken data

och vilken I/O-enhet som är involverade i överföringen. Under tiden DMA-enheten utför överföringen kan processorn exekvera annan kod och när DMA-enheten är klar får processorn ett avbrott. Då kan processorn ta reda på om överföringen fullbordades och därefter vidta lämpliga åtgärder.[4]

3 Problemformulering

Detta kapitel är uppbyggt enligt följande struktur:

1. Problemdefinition: Beskriver hur projektet skall redovisa en möjlig konstruktion av en timerkomponent i en hypervisor.
2. Problemanalys: Beskriver olika perspektiv på hur timerkomponenten kan fungera och vilka rimliga krav och mål som bör ställas på timerkomponenten.
3. Krav- och målspecifikation: Beskriver vilka krav och mål som ställs på timerkomponenten som exempelvis säkerhet och tidsprecision.
4. Avgränsning.

3.1 Problemdefinition

Efter en beskrivning av konceptet virtualisering kan projektets problem definieras enligt följande: *Hur kan en timer virtualiseras så den framstår som flera timers i en hypervisors syfte?* För att besvara denna fråga består projektet av två delar:

- Generell design: Timerkomponenten designas för en generell hypervisor. Detta innebär att timerkomponentens design inte innehåller något som är konstruerat för specifik hårdvara.
- Fallstudie på ARM-arkitekturen: Fallstudien beskriver hur timerkomponenten kan installeras i en hypervisor (bland annat hur ett timeravbrott simuleras). Detta görs genom att implementera timerkomponenten, och vissa delar av en fiktiv hypervisor som kommunicerar med timerkomponenten. Fallstudien syftar också till att visa vilka hänsynstaganden en hypervisor bör ta vid implementation av en timerkomponent.

3.2 Problemanalys

3.2.1 Säkerhet

Med tanke på att en hypervisor skall förbjuda gästsystem att påverka varandra, måste även timerkomponenten leva upp till detta. Därför måste det undersökas vilka säkerhetshot som finns mot timerkomponenten. Detta görs i avsnitt 4.1.1.

3.2.2 Tidsperspektiv

Timerkomponenten kan designas på flera olika sätt beroende på hur tiden skall räknas från det att hypervisorn och gästsystemen sätter sina motsvarande timers:

- Hypervisorns timer: Hypervisorns timer räknar ned kontinuerligt från det att dess timer startas.
- Gästsystemens timer kan räkna ned bland annat enligt följande sätt:
    - Fördröjt realtidsavbrott med väntande signalering: Från det att gästsystemet sätter sin timer räknar timern ner kontinuerligt och när den når noll signalerar timern hypervisorn. Det hypervisorn gör vid mottagandet av denna signal beror på om avsett gästsystem exekveras för stunden. Om gästsystemet exekverar för stunden levererar hypervisorn ett timeravbrott till aktuell gäst, och i annat fall levereras timeravbrottet då det avsedda

gästsystemet byts in för exekvering.

Anledningen till att ha fördröjd avbrott är att då kan exempelvis gästoperativsystemen endast byta applikationer medan gästsystemet exekverar. Vid omedelbar avbrottsleverans är sannolikheten större att gästoperativsystemen byter applikationer trots att applikationerna inte exekverat.

- Realtidsavbrott: Likadant som i fallet ovan med skillnaden att om avsett gästsystem inte exekverar byts det in omedelbart och kan således behandla sitt timeravbrott omedelbart.

  Denna metod är önskvärd i kritiska tillämpningar där avbrott måste betjänas omedelbart. Som beskrevs i föregående punkt finns det nackdelar med denna metod.

- Virtuell tid: Gästsystemens timertid räknas ned när de exekverar. Den tid gästsystemen förbrukar till att exekvera sina instruktioner kallas i fortsättningen ***exekveringstid***.

Om ett operativsystem skall kunna fylla sitt syfte krävs det att operativsystemet kan dela processorns exekveringstid mellan alla applikationer. För detta syfte används en timer som signalerar när ett byte av applikation skall ske. Då är det önskvärt att timern signalerar när applikationen exekverat den avsatta tiden. Därför designas denna timerkomponent sådant att gästsystemen erbjuds virtuell tidsräkning.

3.2.3 Tidsprecision

Det kan uppstå fall då ett gästsystems timer räknar ned men att gästsystemet inte exekverar. Detta på grund av att hypervisorn och timerkomponenten behöver utföra vissa operationer för att uppfylla system- respektive timervirtualisering. Det vill säga hypervisorn och timerkomponenten kan stjäla exekveringstid från gästsystemen med resultatet att gästsystemens exekveringstid förvrängs. Ett annat fall för tidsförvrängning är då leveranser av avbrott fördröjs vilket gör att reell tid förloras. Det senare fallet kan påverka gästsystemets tillämpning om höga tidskrav på leveranser av realtidsavbrott krävs. På grund av tidsbrist undersöks endast detta senare fall.

3.2.3.1 Mål för precision av realtidsavbrott

Det finns applikationsprogram som behöver högfrekventa timeravbrott. Exempelvis för uppdatering av data eller för utförande av någon sorts kontroll. För att få en uppskattning av vilken tidsprecision timerkomponenten bör ha förs ett resonemang om vad applikationer kan tänkas ha för krav på tidsprecision vid leveranser av realtidsavbrott.

För att göra timerkomponenten användbar i så många tillämpningar som möjligt, görs värstafallsantaganden: En processor med låg klockfrekvens i en tillämpning som har krävande tidsprecisionskrav. En låg klockfrekvens för de flesta tillämpningar idag kan tänkas vara 10 MHz. De enklaste och billigaste processorerna idag som är gjorda för en liten mängd uppgifter exekverar normalt sett över 10 MHz.

Rimligtvis kan de flesta tillämpningar utföras korrekt om ett realtidsavbrott förvrängs med mindre än 0,1 ms. Exempelvis kan uppdatering av data eller regelbundna kontroller utföras med en frekvens på 10 kHz. Som nämndes ovan kan även en timers satta värde användas för tidsrelaterade mätningar, vilka kan kräva hög precision. Därför sätts som mål att ***timerkomponenten*** inte skall orsaka en tidsförvrängning större än 50 µs, från det att timern genererar en avbrottssignal tills dess att gästoperativsystemet börjat exekvera sin avbrottsrutin. Detta mål gäller endast timerkomponenten, där hypervisorns bidrag och kommunikationen med fysiska timern bortses. Detta ger hypervisorn 50 µs att exekvera kod på däremellan, för att uppnå en tidsförvrängning på maximalt 100 µs. Dessa två tidsmål sätts under följande antaganden:

- Processorns klockfrekvens är minst 10 MHz.
- De instruktioner timerkomponenten exekverar som bidrar till tidsförvrängningen tar maximalt 10 klockcykler. Detta får anses som ett generöst antagande och får därför inkludera cachemissar. Exempelvis i processorarkitekturen ARM9 tar de flesta instruktionerna maximalt 3 klockcykler.[6]
- Instruktioner för att kommunicera med fysiska timern bortses.

Dessa antaganden medför således att i värsta fall exekveras en instruktioner på en µs. Alltså kan det tillåtas att maximalt 100 instruktioner exekveras från det att timern genererar en avbrottssignal tills det att gästoperativsystemets avbrottsrutin tar över exekveringen. Då dessa antaganden i praktiken ger de sämsta förutsättningarna för att uppnå kort leveranstid, kan följande antagande göras: Om hypervisorn och timerkomponenten exekverar maximalt 100 instruktioner tillsammans från det att timern genererar en avbrottssignal tills det att gästsystemet påbörjar exekveringen av sin avbrottsrutin, så uppfylls kravet om en tidsförvrängning på maximalt 100 µs. Därför baseras målen för tidsförvrängning på antalet exekverade instruktioner.

3.3 Krav- och målspecifikation

Med tanke på vilka krav en hypervisor skall uppfylla definieras följande krav eller mål på timerkomponenten:

- Säkerhetskrav: Inget gästsystem skall kunna påverka timern sådant att hypervisorn eller något annat gästsystem påverkas.
- Funktionalitetskrav: Timerkomponenten skall fungera på två olika sätt sett ur hypervisorns respektive gästsystemens perspektiv:
  - Hypervisorn: Skall kunna ange en viss mängd tid till timerkomponenten och när denna reella tid passerat skall hypervisorn få ett timeravbrott.
  - Gästsystem: Skall kunna ange en viss mängd tid som resulterar i att när aktuellt gästsystem exekverat denna tid så erhålls ett timeravbrott. Detta innebär att om aktuellt gästsystem byts ut kommer dennes motsvarande timers nedräkning att pausas för att sedan återupptas vid återupptagen exekvering.
- Samtliga operationer på timerkomponenten skall fullbordas i konstant tid enligt ordonotationen.
- Minnesutrymmet timerkomponenten behöver får växa som snabbast linjärt med antalet gästsystem.

Målen för tidsförvrängningar i *implementationen* är:

- Timerkomponenten skall inte exekvera fler än 50 instruktioner från det att fysiska timern genererar en avbrottssignal tills dess att gästoperativsystemet börjat exekvera sin avbrottsrutin.
- Hypervisorn skall inte exekvera fler än 50 instruktioner under samma tidsperiod som angivits i punkten ovan.

Dessa två mål tar ej hänsyn till timerspecifik kommunikation.

3.4 Avgränsning

Projektet har följande avgränsningar:

- Generell design:
  - Designen av timerkomponenten tar inte hänsyn till uppgifter som ligger i hypervisors ansvar. Bland annat gäller detta säkerhetshot som skall motverkas av hypervisorn.
  - Timerkomponenten redovisas med ett gränssnitt bestående av funktionsanrop implementerade i C.
  - Timerkomponenten designas för datorer med en exekveringskärna.
- Fallstudie på ARM-arkitekturen:
  - Koden för timerkomponenten, och delar av hypervisorn, skrivs i C respektive assembler. Denna kod testas ej då syftet med fallstudien är att illustrera hur en timerkomponent kan konstrueras och installeras i en hypervisor.
  - För att illustrera hur timerkomponenten kan instlleras i en hypervisor antas en fiktiv hypervisor finnas. Denna hypervisor modifieras för att förenkla placeringen av timerkomponenten.
  - För att förenkla implementationen görs ett antal antaganden angående den fiktiva hypervisorn och ARM-processorns funktionalitet. Dessa antaganden beskrivs vidare i avsnitten 5.1 och 5.2.
  - Specifik timerkommunikation ersätts med kommentarer eller funktionsanrop.

4 Timervirtualisering för hypervisors

4.1 Designanpassning efter krav och mål

4.1.1 Säkerhetshot och motsvarande motmedel

Då timerkomponenten inte får manipuleras hur som helst bör det identifieras hur timerkomponenten kan manipuleras. Detta för att identifiera motmedel mot dessa säkerhetshot. Således kan motmedel hittas mot otillåten manipulering. Med dessa hot och motmedel kända kan det avgöras vilka motmedel timerkomponenten skall implementera och vilka som ligger i hypervisorns ansvar.

I avsnitt 2.4 beskrevs bland annat datorns tre delkomponenter (CPU, minne och I/O). Dessa tre komponenter ger upphov till otillåten manipulering av timerkomponenten och därför beskrivs hur dessa komponenter kan påverka timern samt hur hoten kan motverkas.

4.1.1.1 Processorns privilegierade instruktioner

De hot som kan uppstå med ursprung i processorn är att program som exekveras i det icke-privilegierade läget innehåller instruktioner som är privilegierade. Om ett sådant fall uppstår sker en kontrollövergång. Detta betyder att en övergång sker från det icke-privilegierade läget till det privilegierade läget och att processorn exekverar specifik kod i minnet som svar på detta.

Dessa kontrollövergångar kan hypervisorn använda för att förhindra gästsystem från att ändra hårdvarans konfiguration. Detta gäller inte endast minne och I/O, utan även processorns statusregister. Om statusregisterna kan ändras kan detta resultera i gästsystemen exempelvis kan exekvera i privilegierat läge.

Om processorn är konstruerad sådant att det inte sker kontrollövergångar för *alla* exekverade privilegierade instruktioner i icke-privilegierat läge (vilket är fallet för Intels instruktionsuppsättning IA-32), måste hypervisorn kontrollera varje instruktion ett gästsystem önskar exekvera. I detta fall skulle interpretation vara en lösning. Detta skulle dock sänka prestandan vilket gör att paravirtualisering är kan vara aktuellt.

4.1.1.2 Minnesåtkomster

Ett säkerhetshot som uppstår för gästsystemen angående minnet är att gästsystemen skulle kunna läsa eller skriva över varandras minnesceller. Eftersom hypervisorn ansvarar för att skydda hårdvaran är det dennes ansvar att konfigurera minnet sådant att gästsystem inte kan påverka varandra.

Detta hot minnet ger upphov till kan lösas med en MMU vilken erbjuder virtuellt minne. MMU:n har då ett adressutrymme för varje gästsystem som endast det gästsystemet har åtkomst till. Om ett annat gästsystem försöker skriva eller läsa dessa minnesceller sker en kontrollövergång till hypervisorn, som då behandlar ett sådant undantagsfall.

Ytterligare ett hot som involverar minnet är DMA-enhetens möjlighet att skriva i minnet. DMA-enheten skulle kunna få instruktioner av ett gästsystem att lagra data på en viss plats i minnet. Denna plats i minnet får således inte tillhöra annat gästsystem eller hypervisorn. Detta hot är av samma karaktär som det under avsnitt 4.1.1.1 och motverkas således på samma sätt.

4.1.1.3 Drivrutiner

De drivrutiner hypervisorn har för att sköta kommunikation med I/O-enheterna kan vara ett hot mot timern. Då drivrutinerna är gjorda av externa parter (vars pålitlighet kan vara tvivelaktig) och att de exekverar i privilegierat läge kan detta öppna för säkerhetshot mot gästsystemen. Drivrutinerna skulle exempelvis kunna innehålla möjligheter för vissa indataparameterar manipulera hårdvaran på ett otillåtet sätt. Drivrutinerna skulle till och med kunna vara konstruerade sådant att de manipulerar komponenter i datorns hårdvara som tillverkaren ej angett.

För att motverka detta hot skulle en timerkomponent kunna innehålla en delkomponent bestående av kontrollkod. Denna kontrollkod kan användas av hypervisorn för att kontrollera att inga drivrutiner manipulerar den fysiska timern. Dock har drivrutinerna möjlighet att påverka annat än timern, som exempelvis minnet, och således är även detta hot av liknande karaktär som det under 4.1.1.1.

4.1.1.4 Interna brister i timerkomponenten

Bortsett från hårdvaran, om timerkomponenten skall kunna agera felaktigt krävs det antingen att den innehåller buggar, eller att det går att manipulera den på ett ospecificerat sätt via parametervärden i dess gränssnitt. Dessa hot måste dels motverkas av timerkomponentens design, och dels genom att hypervisorn använder timerkomponenten enligt dess specifikation.

4.1.1.5 Slutsats

Av de identifierade säkerhetshoten är det endast timerkomponentens interna brister skall beaktas vid designen av timerkomponenten. Detta då resterande hot skall motverkas av hypervisorn i och med att hypervisorn ansvarar för att hantera och skydda hårdvaran.

4.1.2 Funktionalitet och minnesutrymme

Då det är gästsystemens exekveringstid timern skall räkna resulterar detta i att ett gästsystems timer endast räknar ned när gästsystemet exekverar. När gästsystemet byts ut pausas dennes timer, och när nästa gästsystem byts in startas detta gästsystems timer. Denna form av timervirtualisering kan uppnås enligt följande princip (beskriven i detalj i avsnitt 4.2.2). Hypervisorn och varje gästsystem har en egen minnescell. Gästsystemens minnescell innehåller hur mycket exekveringstid de har kvar tills de skall få ett timeravbrott. Hypervisorns minnescell beskriver hur mycket reell tid denne har kvar till sitt timeravbrott. När timern startas sätts den till det värde som är minst av hypervisorns och aktuellt exekverande gästsystems timervärde, annars fördröjs avbrott. Utöver detta behöver timerkomponenten ett antal variabler, exempelvis ID:t på aktuellt exekverande gästsystem.

Således uppfylls kraven angående funktionalitet, och att minnesutrymmet växer linjärt med antalet gästsystem.

4.1.3 Tidsprecision och operationer i Ordo-konstant tid

Som beskrevs i avsnitt 3.2.3 orsakar hypervisorn och timerkomponenten tidsförvrängningar. Dessa tidsförvrängningar medför följande konsekvenser för gästsystemen:

- Då ett gästsystems timer kan räkna ned utan att gästsystemet exekverar innebär detta att ett gästsystem kan få en skev uppfattning av sin exekveringstid.
- När en timer räknat ned till noll genereras ett avbrott i processorn. Då börjar operativsystemet exekvera en rutin som svar på detta. I fallet med systemvirtualisering

fördröjs exekveringen av denna rutin vilket gör att gästsystemens behandling av sina timeravbrott tidsförskjuts. Således förvrängs gästsystemens reella tid vid behandling av realtidsavbrott.

För att minska dessa negativa påverkningar på gästsystemen identifieras felkällorna.

4.1.3.1 Felkällor

Fallen nedan har identifierats som möjliga felkällor till att gästsystemens exekveringstid eller realtidsuppfattning förvrängts i samband med att timern manipulerats eller att ett avbrott uppstått:

- **Timern startas:** Efter att hypervisorn startat timern exekverar hypervisorn ett antal instruktioner för att lämna över exekveringen till aktuellt gästsystem. Detta resulterar i att timern startar sin nedräkning innan gästsystemet påbörjat sin exekvering. ***Alltså förlorar gästsystemet exekveringstid på grund av hypervisorn.***
- **Timern pausas:** När gästsystemet lämnar över exekveringen till hypervisorn kräver detta exekvering av ett antal instruktioner innan timern pausas. Således kan timern räkna ned under tiden hypervisorn exekverar och innan timern pausas. ***Alltså förlorar gästsystemet exekveringstid på grund av hypervisorn.***
- **Följande fall kan vara aktuella då timeravbrott sker:**
  a) **Generellt:** När ett avbrott sker pausas den normala exekveringen omedelbart, oavsett vem avbrottet är till (hypervisorn eller aktuellt gästsystem). Detta gör att ett gästsystem inte kan exekvera färre eller flera instruktioner än i fallet utan hypervisor. ***Alltså påverkas inte exekveringstiden av timeravbrott.***
  b) **Realtidsavbrott till gästsystem då hårdvaran ej har stöd för virtualisering:** Då hypervisorn bland annat måste lämna över exekveringen till gästsystemet kräver detta att hypervisorn exekverar instruktioner innan gästoperativsystemet kan behandla sitt timeravbrott. Således fördröjs exekveringen av gästoperativsystemets avbrottsrutin. ***Alltså förlorar gästsystemet reell tid på grund av hypervisorn.***
  c) **Realtidsavbrott till gästsystem:** Innan ett timeravbrott kan levereras till aktuellt gästsystem måste hypervisorn interagera med timerkomponenten, bland annat för att identifiera vem avbrottet är till: hypervisorn eller aktuellt gästsystem. Resultatet är att gästsystemet får sitt timeravbrott fördröjt. Detta påverkar inte exekveringstiden men det påverkar leveranstiden av realtidsavbrott. ***Alltså förlorar gästsystemet reell tid på grund av timerkomponenten.***
- **Aktuellt gästsystem byts ut då hypervisorns timer når noll:** I detta fall antas det att hypervisorn detekterar en signal från timern som tolkas som att aktuellt gästsystem skall bytas ut. När processorn detekterar signalen pausas gästsystemet omedelbart, och byts ut. Således kan inte gästsystemet exekvera mer eller mindre sett till dennes timertid. ***Alltså förloras ingen exekveringstid.***
- **Nytt gästsystem byts in:** När hypervisorn byter in ett nytt gästsystem måste hypervisorn sätta och starta timern innan exekveringen av gästsystemet kan påbörjas. Under överlämning av exekveringen till nya gästsystemet kommer gästsystemet förlora exekveringstid. Tidsförvrängningen i detta fall orsakas av samma anledning som i fall 1. ***Alltså förloras exekveringstid på grund av hypervisorn.***
- **Repetition av uppgift:** Om ett gästsystem exekverar under en viss tid, och under denna tid får ett antal avbrott, kommer detta leda till att en viss mängd tid förloras enligt vad som beskrivits i punkterna ovan. Denna förlorade tid skulle kunna resultera i att gästsystemet inte

hinner fullborda en viss tidsberoende uppgift innan gästsystemet byts ut. En sådan uppgift skulle kunna vara att upprätta en kommunikationslänk med en annan dator där en time-out sker, vilket resulterar i att anslutningen bryts. Resultatet kan således bli att denna uppgift måste göras om och därmed har allt arbete som gjorts innan varit förgäves. Således kan även gästsystemen förlora exekveringstid på detta sätt. Detta beror dock på tidsförvrängningarna ovan. ***Alltså förlorar gästsystemet exekveringstid på grund av hypervisorn och timerkomponenten.***

4.1.3.2 Slutsats

I vissa fall kan det tänkas att timerkomponenten måste manipuleras efter fysiska timern startas eller före den pausas. Om sådan manipulation är nödvändig orsakar timerkomponenten tidsförvrängningar. Således är det önskvärt att i sådana fall pausa timern så fort som möjligt eller starta den så sent som möjligt. Därmed erhålls följande slutsats:

- Hypervisorn orsakar förvrängning av exekveringstiden i samband med att timern startas och pausas. Därför bör manipulering av timerkomponentens ske innan eller efter fysiska timern startas respektive pausas, samt att överlämningen av exekveringen till gästsystemen görs så effektiv som möjligt.
- Timerkomponenten, och eventuellt hypervisorn, orsakar realtidsförskjutning vid leveranser av timeravbrott till gästsystemen. Således bör:
    1. Interaktionen mellan hypervisor och timerkomponenten minimeras i samband med avbrott. Det vill säga minimera antalet interaktioner, och där varje sådan utförs med få instruktioner.
    2. Ett litet antal instruktioner exekveras när hypervisorn lämnar över exekveringen till aktuellt gästsystems avbrottsrutin.

Manipulering av timerkomponentens data rör inte uppgifter som beror på antalet gästsystem, eller storleken på minnescellernas timervärden. Därmed kan alla operationer på timerkomponenten utföras i konstant tid enligt Ordonotationen.

4.2 Timerkomponentens design

4.2.1 Timerkomponentens gränssnitt

Det som påverkar timerkomponentens gränssnitt är vad gästsystemen vill ha för funktionalitet av en fysisk timer, och vad en hypervisor behöver för funktionalitet från timerkomponenten. Gästsystemen förväntar sig samma funktionalitet som en fysisk timer har: Sätta timertid, läsa timertid, starta nedräkning, pausa nedräkning och nollställning av timern. För hypervisorn visar timerkomponenten två timers: En för gästsystemen och en för just hypervisorn. Utöver de funktioner gästsystemen behöver av en fysisk timer, behöver hypervisorn ange vilket gästsystem som exekverar för tillfället. Detta måste hypervisorn ange för timerkomponenten så att rätt gästsystems exekveringstid räknas ned.

För enkelhetens skull kan läsaren tänka sig att timerkomponenten innehåller en timer för varje användare, istället för två timers enligt principbeskrivningen i avsnitt 4.1.2. Således får hypervisorn och varje gästsystem en egen timer. Följande funktioner definieras av timerkomponenten (vilka finns implementerade i C-kod i Appendix A):

- *void startTimer()*: Den fysiska timern startas med aktuellt innehållande värde.

- *void pauseTimer()*: Den fysiska timern pausar sin nedräkning.
- *void stopHypervisorsTimer()*: Nollställer hypervisorns timer.
- *void stopCurrentVMsTimer()*: Nollställer aktuell gästs timer.
- *void setHypervisorsTimer(int newTimerValue)*: Hypervisorns timervärde sätts till *newTimerValue*.
- *void setCurrentVMsTimer(int newTimerValue)*: Aktuell gästs timervärde sätts till *newTimerValue*.
- *int getHypervisorsTimerValue()*: Returnerar mängden tid som återstår tills hypervisorns timer räknat ned till noll.
- *int getCurrentVMsTimerValue()*: Returnerar mängden exekveringstid som återstår tills aktuell gästs timer räknat ned noll.
- *void changeCurrentVM(int newVM)*: Byter exekverande gäst. Nya gästen har ID-värdet *newVM*, där $0 \leq newVM < antal\ gästsystem$.
- *int updateTimerComponent()*: Uppdaterar variabler och eventuellt värdet i fysiska timern i samband med att ett timeravbrott skett. Denna funktion är involverad i realtidsavbrott och bör därför optimeras. Om avbrottet är till aktuell gäst och hypervisorn har ett timervärde större än noll, avslutar denna funktion med ett anrop till *startTimer()*. Detta för att undvika onödigt anrop till *startTimer()*. Funktionen returnerar följande värden beroende på vem avbrottet är till:

  1. Avbrott till hypervisorn.
  2. Avbrott till aktuellt gästsystem.
  3. Avbrott till båda.

  Hypervisorn måste anropa denna funktion för att veta vem avbrottet är till, och därmed tvingas hypervisorn att uppdatera timerkomponenten på detta korrekta sätt. Därmed kan inte timerkomponenten manipuleras inkorrekt vid ett avbrott.
- *void initTimerComponent(int currentVM, int numberOfVMs)*: Initierar timerkomponentens datavariabler: Alla användare får noll som timervärde, *currentState* får värdet 6 och aktuell gäst har ID-värdet *currentVM*. Timerkomponenten är efter detta redo att användas.

Detta gränssnittet kräver att timerkomponenten innehåller all relevant timerdata, detta då de ej bifogas som parametrar i funktionerna. Vidare kräver timerkomponenten, för korrekt exekvering, att fysiska timern är paused medan funktionerna exekverar. Därför skall hypervisorn pausa timern innan timerkomponenten manipuleras, och därefter eventuellt starta timern. Resultatet är att tidsförvrängningen av gästsystemens exekveringstid minimeras.

Angående säkerheten i timerkomponentens gränssnitt är det två saker som bör beaktas: Vilka är de nödvändiga parametrarna, och vilka värden dessa parametrar skall ha. Först och främst skall hypervisorn använda timerkomponenten som den är specificerad att användas: *pauseTimer()* anropas innan timernkomponenten manipuleras, utom vid avbrott då *getinterruptcase()* och *updateTimerComponent()* anropas i denna ordning. Sedan angående parametrarna är det timervärdena och aktuell gästs ID-värde som skall vara korrekta. Timervärdenas tillåtna storlek beror på den fysiska timern och därför lämnas denna kontroll utanför denna timerkomponent och därför också negativa värden. Vid en implementation bör kontrollen göras inom timerkomponenten för tydlig kodstruktur. Angående ID-värdet på aktuell gäst är det hypervisorns ansvar att kontrollera att det är korrekt, då det är hypervisorn som hanterar generell information angående gästsystemen.

4.2.2 Design av timerkomponentens funktionalitet

Timerkomponentens specifikation angående funktionaliteten är enligt följande:

- Hypervisorns timer skall räkna ned kontinuerligt, det vill säga räkna reell tid.
- Ett gästsystems timer skall endast räkna ned medan det exekverar, det vill säga räkna exekveringstid.

För att förstå timerkomponentens konstruktion bör läsaren tänka sig att hypervisorn och varje gästsystem har en egen timer. Det finns tre komponenter som skall särskiljas:

- Fysiska timerns timervärde: Hur lång tid det är kvar tills fysiska timern genererar ett avbrott.
- Hypervisorns timervärde: Hur lång tid det är kvar tills hypervisorn skall få ett timeravbrott.
- Respektive gästsystems timervärde: Hur lång tid det är kvar av motsvarande gästsystems *exekvering* tills det skall få ett timeravbrott.

För att hålla reda på hypervisorns och gästsystemens timervärden lagras de i minnesceller. Minnescellernas timervärden uppdateras endast vid följande fall:

- När fysiska timern sätts till ett nytt timervärde. Det vill säga när hypervisorn eller aktuellt exekverande gästsystem önskar sätta ett nytt timervärde i deras respektive timers.
- När hypervisorn eller aktuellt exekverande gästsystem önskar pausa eller nollställa sin motsvarande timer.
- När aktuellt exekverande gästsystem byts ut.
- När fysiska timerns räknare räknat ned till noll. Det vill säga när timern genererar ett avbrott.

För att förstå hur dessa minnesceller skall hanteras bör tre aspekter beaktas:

1. Hypervisorns timervärde kan ändras när som helst. Detta då hypervisorns timer skall räkna ned kontinuerligt.
2. Ett gästsystems motsvarande minnescell ändrar endast värde när just det gästsystemet exekverar. Detta då gästsystemens timer endast skall räkna ned vid deras respektive exekvering.
3. När fysiska timern sätts är det alltid den minnescell med minst timervärde som används. De två timervärden som jämförs tillhör hypervisorns respektive aktuellt gästystems minnesceller. Om inte det minsta värdet sätts i fysiska timern fördröjs den andre användarens timeravbrott.

Därmed kan slutsatsen dras att *minnescellerna innehåller hur mycket tid det är kvar tills det att dess motsvarande användare skall få ett avbrott, relativt senaste tillfället fysiska timerns värde sattes.*

Således kan det ses att det finns ett samspel mellan dessa timervärden: Fysiska timerns timervärde, hypervisorns timervärde och aktuellt exekverande gästsystems timervärde. För att timerkomponentens funktioner skall agera korrekt krävs det att dessa funktioner är medvetna om dessa tre timervärden. Främst beror funktionerna på hur hypervisorns och aktuellt gästsystems timervärden förhåller sig till varandra. Detta då de implicerar vilket timervärde som sattes i den fysiska timern, det minsta av dem. Därför kan timerkomponenten befinna sig i ett antal tillstånd, där varje tillstånd beror på hur dessa två timervärden förhåller sig till varandra.

4.2.2.1 Möjliga tillstånd

Då flera funktioner beror på vilket tillstånd timerkomponenten befinner sig i, definieras koder som beskriver vilket tillstånd timerkomponenten befinner sig i. Koderna består av heltal som motsvarar följande tillstånd:

1. Hypervisorn har ett timervärde som är mindre än gästsystemets timervärde. Därför är det hypervisorns timervärde som sattes i fysiska timern när denne startades senast.
2. Hypervisorn har ett timervärde som är större än gästsystemets timervärde. Därför är det gästsystemets timervärde som sattes i fysiska timern när denne startades senast.
3. Endast Hypervisorn har ett timervärde. Fysiska timern påverkas på samma sätt som i fall 1.
4. Endast gästsystemet har ett timervärde. Fysiska timern påverkas på samma sätt som i fall 2.
5. Hypervisorn och aktuellt gästsystem har lika stora timervärden.
6. Varken Hypervisorn eller gästen har timervärde. Fysiska timern är därför i passivt läge.

Om en minnescell innehåller värdet noll betyder detta att användaren av denna minnescell ej använder sin timer.

4.2.2.2 Avbrott

Hittills har det beskrivits hur timerkomponenten sätter ett timervärde och hanterar det. I detta avsnitt beskrivs hur timerkomponenten tar hand om avbrott. Det finns tre fall beroende på vem/vilka avbrottet är till. Dessa upptäcks och behandlas enligt följande:

1. Avbrott till hypervisorn: Om aktuellt tillstånd är 1 eller 3 betyder detta att hypervisorn har minst timervärde eller att endast hypervisorn använder sin timer. Alltså är avbrottet till hypervisorn. Detta resulterar i att programmet går till hypervisorns avbrottsrutin. Då hypervisorn kontrollerar hela systemet är det upp till denna avbrottsrutin att bestämma vad som skall ske härnäst.
2. Avbrott till aktuell gäst: Om aktuellt tillstånd är 2 eller 4 betyder detta att aktuellt gästsystem har minst timervärde eller att endast aktuell gäst använder sin timer. Alltså är avbrottet till aktuell gäst. Därför lämnar hypervisorn över exekveringen till den rutin gästoperativsystemet exekverar vid timeravbrott.

   Funktionen *updateTimerComponent()* anropas i samband med att avbrott sker. I fallet då gästsystemet får ett avbrott gör denna funktion följande:

   1. Om hypervisorn har ett timervärde, uppdateras detta och sätts i fysiska timern. Annars görs inget av detta.
   2. Aktuellt gästsystems timervärde sätts till noll.
   3. Beroende på om hypervisorn har ett timervärde sätts *currentState* till 3, annars till 6.
   4. Om hypervisorn har ett timervärde, startas timern.
3. Avbrott till båda: Om aktuellt tillstånd är 5 är hypervisorns timervärde lika med aktuell gästs timervärde. Därför skall båda ha avbrott samtidigt. Då hypervisorn är den som bestämmer i systemet och därmed har högre prioritet levereras avbrottet till hypervisorn.

   Då det finns ett väntande avbrott till aktuell gäst är det upp till hypervisorn att bestämma när detta meddelande skall ges till gästen. Exempelvis kanske hypervisorn vill göra ett byte av exekverande gästsystem. I sådant fall kommer avbrottet att meddelas till aktuell gäst när det byts in igen.

4.2.2.3 Timerkomponentens variabler

För att timerkomponenten skall hålla reda på diverse värden används följande variabler:

- *int currentState*: Innehåller timerkomponentens aktuella tillstånd.
- *int hypervisorsTimerValue*: Innehåller hypervisorns timervärde.
- *int[] VMsTimerValue*: En array som innehåller alla gästsystems timervärden. Indexeras med *currentVM*.
- *int currentVM*: Identifierar vilket gästsystem som exekverar för tillfället. $0 \leq currentVM <$ antal gästsystem.

4.2.2.4 Exempelscenario

Ett scenario får ge en djupare inblick i varför denna princip fungerar. I detta scenario antas det att fysiska timerns timervärde specificeras i millisekunder, och att det finns tre gästsystem varav gästsystem 3 exekverar. Scenariot går till enligt följande (se figur 4.1):

1. Fysiska timern är i passivt tillstånd då varken hypervisorn eller gäst 3 använder sina timers och *currentState* har värdet 6.
2. **Hypervisorn pausar timern och sätter sin timer till 4 sekunder:** Först kontrolleras timerkomponentens tillstånd genom att läsa *currentState*. c*urrentState* har värdet 6 vilket betyder att ingen använder timern. Därför lagras värdet 4000 lagras i hypervisorns minnescell och i timern, och *currentState* sätts till 3. **Hypervisorn startar timern.**
3. **Två sekunder senare pausar hypervisorn fysiska timern då aktuell gäst sätter sin timer till en sekund:** Aktuellt tillstånd kontrolleras vilket visar att endast hypervisorn har ett timervärde. Fysiska timerns värde läses och dess värde subtraheras från hypervisorns minnescell: 4000 – 2000 = 2000. Aktuell gästs minnescell uppdateras till värdet 1000. En jämförelse sker mellan aktuell gästs timervärde och hypervisorns timervärde. Då aktuell gäst har ett mindre timervärde lagras detta i fysiska timern och *currentState* sätts till 2. **Hypervisorn startar timern.**
4. **En sekund senare uppstår ett avbrott:** Hypervisorn kontrollerar vem/vilka avbrottet är till genom att läsa variabeln *currentState*. Denna variabel har värdet 2 vilket betyder att avbrottet är till aktuell gäst. Enligt fall 2 i avsnitt 4.2.2.2 subtraheras hypervisorns timervärde med aktuell gästs timervärde: 2000 – 1000 = 1000. Detta resultat motsvarar hur mycket tid hypervisorn har kvar till sitt avbrott och sätts därför i fysiska timern. Sedan sätts aktuell gästs timervärde till 0 då denne inte har något timervärde, och *currentState* sätts till 3. **Fysiska timern startas och hypervisorn lämnar över exekveringen till den rutin i gästoperativsystemet som hanterar timeravbrott.**
5. **En halv sekund senare pausar hypervisorn timern och nollställer sin timer:** Då *currentState* har värdet 3 betyder detta att endast hypervisorn har ett timervärde. Därför sätts hypervisorns minnescell till noll och *currentState* till 6.

Efter steg 1

Användare	Värde
Hypervisor	0
Gäst 1	750
Gäst 2	0
Gäst 3	0
Aktuell gäst	3
Aktuellt tillstånd	6
Fysiska timern	0

Efter steg 2

Användare	Värde
Hypervisor	4000
Gäst 1	750
Gäst 2	0
Gäst 3	0
Aktuell gäst	3
Aktuellt tillstånd	3
Fysiska timern	4000

Efter steg 3

Användare	Värde
Hypervisor	2000
Gäst 1	750
Gäst 2	0
Gäst 3	1000
Aktuell gäst	3
Aktuellt tillstånd	2
Fysiska timern	1000

Efter steg 4

Användare	Värde
Hypervisor	1000
Gäst 1	750
Gäst 2	0
Gäst 3	0
Aktuell gäst	3
Aktuellt tillstånd	3
Fysiska timern	1000

Efter steg 5

Användare	Värde
Hypervisor	0
Gäst 1	750
Gäst 2	0
Gäst 3	0
Aktuell gäst	3
Aktuellt tillstånd	6
Fysiska timern	0

Figur 4.1: Exempelscenariots påverkan på timerkomponentens variabler och på fysiska timerns värde. Notera att de övriga gästernas timervärden inte påverkas.

5 Implementation av timervirtualisering

I denna fallstudie kan läsaren tänka sig ett scenario där en timerkomponent skall implementeras i en (fiktiv) hypervisor, som körs på smart phones. Exempelvis skulle en sådan hypervisor kunna användas för att köra Android eller iOS på samma telefon. Därmed kan användaren använda det operativsystem som denne känner för när som helst.

Fallstudien har två syften:

- Illustrera hur timerkomponenten, utvecklad i föregående avsnitt, kan installeras i en hypervisor. Detta görs genom att beskriva följande fall:
  - Hur minnet i en hypervisor kan användas för att erbjuda säkra minnesceller till timerkomponenten.
  - Hur ett avbrott uppstår och levereras till aktuellt gästsystem.
  - Hur aktuellt gästsystem sätter sin timer.
- Visa vilka hänsynstaganden en hypervisor kan behöva göra för att implementera en timerkomponent.

Det som görs i detta scenario ses ur timerkomponentens perspektiv där ett antal antaganden görs för att förenkla lösningarna men ändå sådant att de viktigaste aspekterna framgår som bör beaktas vid timervirtualisering. Annat som hypervisorn måste kunna göra, som exempelvis byta ut exekverande gästsystem, tas ej hänsyn till vid denna implementation. Det vill säga timerkomponenten implementeras ej för att fungera på ett smidigt sätt ihop med en hypervisors övriga komponenter.

5.1 En fiktiv hypervisor

Detta avsnitt beskriver den fiktiva hypervisorn. Läsaren kan tänka sig att ett gästoperativsystem exekveras nästintill kontinuerligt. Endast när ett gästsystem önskar manipulera hårdvaran tar hypervisorn över kontrollen temporärt. Under denna tid manipulerar hypervisorn den del av hårdvaran som är tilldelad aktuellt gästsystem. När hypervisorn är färdig med denna uppgift återlämnas exekveringen till gästsystemet. Gästsystemet återupptar då exekveringen från den punkt där där hypervisorn tog över kontrollen.

För att inte gästsystemen skall kunna göra vad de vill måste de exekveras i icke-privilegierat läge. Då operativsystem i normala fall exekverar i privilegierat läge måste de modifieras för att kunna fungera korrekt under exekvering i icke-privilegierat läge. Detta betyder att operativsystemen paravirtualiseras vilket gör dem till gästsystem. Paravirtualiseringen i detta fall innebär att varje gång gästsystemen vill manipulera hårdvaran, anropar de ett så kallat hypercall. Hypercalls är en form av funktionsanrop som signalerar till hypervisorn vad aktuellt gästsystem vill ha för hårdvarubetjäning.

Om undantag eller avbrott sker tar hypervisorn också över kontrollen. Sedan simuleras dessa undantag och avbrott genom att manipulera gästoperativsystemets register/variabler. Resultatet blir då att ett gästsystem uppfattar att det orsakat ett undantag eller fått ett avbrott på samma sätt som i fallet utan hypervisor. Därefter kan gästoperativsystemet hantera undantaget eller avbrottet.

Om användaren önskar byta operativsystem sköter hypervisorn detta. Ett byte av operativsystem görs genom att hypervisorn lagrar undan aktuellt gästsystems alla register till gästsystemets motsvarande datastruktur. Varje gästsystem har en datastruktur i hypervisorns minne som innehåller relevant exekveringsdata för just det gästsystemet. Sedan återställs det nya gästsystemets register

och exekveringen av det nya gästsystemet kan börja.

All dessa tilltag görs för att gästsystemen inte skall kunna påverka varandra.

5.2 ARM-arkitekturen

ARM är en välanvänd processorarkitektur i inbyggda system, däribland smartphones. ARM är ett företag som designar processorer utan att tillverka dem. Information angående ARM-arkitekturen finns i[6].

Antaganden angående den fiktiva hypervisorn beskrevs ovan. Antagandena angående ARM-processorn är följande:

- 32-bitars arkitektur vilket är standard i ARM.
- Innehar en MPU för att erbjuda skydd av minnet. Oftast innehas en MMU med virtuellt minne men för att göra illustrationen enklare används en MPU.
- MMIO: Den vanligaste förekommande tekniken för kommunikation med I/O-enheter i ARM-arkitekturen.

5.2.1 Exekveringslägen

ARM-processorerna kan exekveras i sju olika lägen. Det finns ett icke-privilegierat läge kallat user. Det är i detta läge processorn befinner sig i när applikationer exekverar (eller i denna fallstudie gästsystem). Sex av lägena är privilegierade varav de nedan fem första är undantagslägen (indantagslägen beskrivs i avsnitt 5.2.4):

- Abort: Processorn hamnar i detta läge då en minnesåtkomst nekas.
- Fast interrupt request: När ett avbrott av högre prioritet sker hamnar processorn i detta läge.
- Interrupt request: När ett avbrott av lägre prioritet sker hamnar processorn i detta läge.
- Supervisor: Det läge processorn startar i och som ett operativsystem normalt exekverar i.
- Undefined: Det läge processorn hamnar i om en icke definierad instruktion skall exekveras.
- System: Har åtkomst till samma register som user-läget med skillnaden att statusregistret CPSR kan skrivas.

5.2.2 Register

Det exekveringsläge processorn befinner sig i definierar vilka register som finns åtkomliga i aktuell exekvering. Det finns 18 olika sorters register: 16 generella register benämnda r0-r15 samt två ytterligare register CPSR och SPSR. De tre sista generella registerna samt CPSR och SPSR har följande roller:

- r13: SP, stackpekare. Innehåller den adress som innehåller det senaste elementet som är lagt på stacken.
- r14: LR, länk-register. Innehåller återhoppsadressen vid funktionsanrop.
- r15: PC, programräknaren: Innehåller adressen till den minnescell som innehåller nästa instruktion som skall hämtas.
- CPSR, Current Program Status Register: Innehåller aktuellt exekveringsläge processorn

befinner sig i, samt status- och kontrollflaggor. Kan endast manipuleras då processorn är i privilegierat läge. Detta gör att user-läget inte kan ändra exekveringsläget.

- SPSR, Saved Program Status Register: När ett undantag sker lagras CPSR i SPSR.

En del av dessa 18 register finns i flera uppsättningar som är oberoende av varandra:

- System- och user-läget har samma register utom SPSR som inte finns tillgängligt i dessa två exekveringslägen.
- De resterande fem exekveringslägena har bland annat egna uppsättningar av SP-, LR- och SPSR-registerna.
- CPSR delas av alla exekveringslägen.

5.2.3 Programräknaren, PC

För att förstå hur ett avbrott skall behandlas bör pipelinen beskrivas samt vektortabellen och undantag. Pipelinen är den del av ARM-processorn som bearbetar instruktioner. I fallstudien antas denna bestå av tre faser (vissa ARM-processorer har fem) där endast en instruktion kan befinna sig i respektive fas. För varje tidssteg förflyttas en instruktion till nästa fas:

1. Hämtning: Den instruktion som finns på den minnesadress programräknaren innehåller hämtas.
2. Avkodning: Identifiering av aktuell instruktion för att kunna exekvera den i nästa steg.
3. Exekvering: Den instruktion som är i denna fas är den som håller på att exekveras. När denna fas är över kommer instruktionen att ha fått sin verkan.

Denna process gör att programräknaren manipuleras enligt följande:

1. Instruktion A som programräknaren pekar på hamnar i hämtningsfasen.
2. Programräknaren ökar med 4 (i fallet med 32-bitars processor blir det 4 byte till). Instruktion A flyttas till avkodningsfasen och instruktion B hämtas enligt programräknarens värde.
3. Programräknaren ökar med 4. Instruktion A hamnar i exekveringsfasen, instruktion B i avkodningsfasen och instruktion C hämtas enligt programräknarens värde.

Låt PC beteckna instruktion A:s minnesadress. Om tiden fryses direkt efter steg 3 har instruktion A precis exekverats och programräknaren kommer ha sitt ursprungliga värde adderat med åtta. Instruktion B som är nästa instruktion att exekveras har dock minnesadress PC + 4. Programräknaren har ett värde som är 4 byte mer än nästa instruktion som skall exekveras. Detta är viktigt att tänka på när den vanliga exekveringen skall fortsätta efter ett undantag så att inte en instruktion försvinner ur den vanliga exekveringen.

5.2.4 Undantag och Vektortabellen

Följande beskriver vad som sker i processorn när ett undantag uppstår, vilket inkluderar avbrott:

1. Den instruktion som håller på att exekveras slutförs.
2. Om aktuellt undantag gör att processorn ändrar exekveringsläge, vilket sker om aktuellt exekveringsläge är user, sparas CPSR och PC i motsvarande undantagsläges SPSR respektive LR. Om exekveringsläget inte ändras på grund av undantaget lagras inte CPSR och PC i CPSR och LR.

   Det finns dock undantagsfall som inte gör på detta sätt. Bland annat vid mjukvaruavbrott

lagras PC – 4 i LR.

SIST sätts CPSR till ett värde som representerar det nya exekveringsläget.

3. Programräknaren sätts till det värde i vektortabellen som motsvarar aktuellt undantagsfall och därefter rensas pipelinen.
4. Instruktionen som finns på motsvarande plats i vektortabellen exekveras (appendix A.12 visar hur en vektortabell kan se ut). Denna instruktion resulterar, i de flesta fall beroende på implementation och undantagsfall, i att programräknaren pekar på starten av den rutin som behandlar aktuellt undantagsfall.

När rutinen som behandlar aktuellt undantagsfall är färdigexekverad skall den vanliga exekveringen återupptas. Innan den vanliga exekveringen återupptas skall registerna återställas. De sista registerna som skall återställas är CPSR och programräknaren. Detta görs genom att CPSR sätts till värdet i SPSR, och programräknaren sätts till värdet i LR (vilket skall subtraheras i vissa undantagsfall). Efter att programräknaren satts till LR återupptas den vanliga exekveringen.

5.3 Minneskonfiguration för ARM-processor med MPU

Denna del illustrerar vad som måste göras för att skydda timerkomponentens data i minnet.

För att skydda timerns data i minnet antas att ARM-processorn använder en MPU, Memory Protection Unit. En MPU kan dela upp minnet i olika regioner där respektive minnesregion får specifika åtkomsträttigheter. Vilken åtkomsträttighet som skall gälla vid en specifik minnesåtkomst baseras på processorns exekveringsläge. I denna fallstudie delas minnet upp i två regioner: En till hypervisorn, och en till aktuellt exekverande gästsystem. Då MPU:n endast kan konfigureras i privilegierat läge kan inte gästsystemen ändra en minnesregions åtkomsträttigheter.

För att skydda timerkomponentens data läggs den i hypervisorns minnesregion. Då I/O-enheter tillhör hypervisorn och MMIO används**,** placeras I/O-enheternas motsvarande minnesceller i hypervisorns minnesregion.

Minnet antas vara 1 GB stort och delas upp enligt följande:

- Hypervisorns minnesområde: Startar på adress 0x0 och slutar på adress 0x3FF, det vill säga 1 MB stort. En lämplig storlek beror på hur mycket kod hypervisorn har. Denna region har endast privilegierat exekveringsläge åtkomst till, både för läsning och skrivning.
- Gästsystemens minnesområde: Startar på adress 0x400 och slutar på adress 0x3FFFFFFF, det vill säga resterande minne. Denna minnesregion har både privilegierat och icke-privilegierat läge åtkomst till för både läsning och skrivning. Anledningen till att hypervisorn skall kunna skriva i denna region är för att hypervisorn lagrar undan gästsystemets register på dess stack, vilket beskrivs i avsnitt 5.5.1.

5.3.1 ARM-processorns MPU

5.3.1.1 Minneskonfiguration

MPU:n konfigureras via en koprocessor. En koprocessor är en mindre processor som kan utföra vissa uppgifter åt CPU:n. I ARM-processorernas fall är det koprocessor 15 (i fortsättningen kallad CP15) som sköter konfigurationen av MPU:n. CP15 kan programmeras via specifika instruktioner som exekveras på CPU:n.

En minnesregions möjliga åtkomsträttigheter visas i tabell 5.1 (kolumnen Kod kan ignoreras).

Privilegierat läge	Icke-privilegierat läge	Kod
No access	No access	0b00
Read/Write	No access	0b01
Read/Write	Read	0b10
Read/Write	Read/Write	0b11

Tabell 5.1: Minnesregionernas möjliga åtkomsträttigheter beroende på processorns exekveringsläge.

CP15 har ett antal primärregister, varav vissa primärregister i sin tur har ett antal sekundärregister. Sekundärregisterna kan ses som utökningar av primärregisterna för att bredda primärregisternas användningsområde. Detta förkortas i sekvensen: CP15:ca:cb, där c är symbolen för ett register i koprocessorn, a är ett tal som identifierar ett primärregister och b ett sekundärregister). Genom att skriva till dessa sekundärregister kan MPU:n konfigureras. Konfigurationen av minnet måste göras enligt följande sekvens:

1. Startadress och storlek konfigureras för respektive region.
2. Rättigheterna konfigureras för respektive region.
3. Regionerna aktiveras.
4. MPU:n aktiveras.

Effekten av detta är att minnet delas upp i olika regioner enligt konfigurationen i steg 1, och att respektive region får åtkomsträttigheterna angivna i steg 2.

5.3.1.2 Minnesregioner

Varje minnesregion identifieras med ett tal mellan noll och sju (inklusive). Detta tal fungerar även som en prioritet för regionen där högre tal ger högre prioritet. Då olika regioner kan överlappa varandra i minnet används dessa prioritetsvärden för att bestämma vilken regions regler som skall gälla. Detta koncept används på följande sätt:

- Region 1 motsvarar exakt hypervisorns minnesregion.
- Region 0 täcker hela minnet och både privilegierat och icke-privilegierat läge har läs- och skrivrättigheter. Delen av denna region som inte tillhör region 1 tillhör exekverande gästsystem.

Dessa regionstal gör att hypervisorns region får högre prioritet än de exekverande gästsystemens region. Därför kan inte gästsystemen påverka hypervisorns region. Notera att denna minneskonfiguration kräver att endast ett gästsystem finns i minnet i taget. Annars skulle det exekverande gästsystemet kunna manipulera ett annat gästsystem.

5.3.1.2.1 Konfiguration av startadress och storlek

Angående regionernas startadress och storlek måste följande två krav följas: Storleken på en region skall vara en tvåpotens vars värde varierar mellan 4 KB och 1 GB, och en regions startadress är en multipel av dess storlek.

För att konfigurera startadress och storlek i region x med prioritet x skrivs ett 32-bitarsvärde till CP15:c6:cx. Register CP15:c6:cx har följande bitfält:

- [31:12] definierar regionens startadress. De lägsta bitarna [11:0] behöver ej beaktas då detta

skulle betyda att en region startar på en adress som inte är en multipel av 4 KB (2^{12}=4096), vilket är minsta tillåtna storlek på en region.

- [11:6] skall vara noll.
- [5:1] definierar regionens storlek. Dessa fem bitar representerar ett decimalt värde n. Regionens storlek blir 2^{n+1}.
- [0] aktiverar konfigurationen i aktuell region om värdet är ett.

Regionernas 32-bitars värden som skrivs till CP15:c6:cx blir därför:

- Region 1: Startadressen är 0 vilket ger bitarna [31:12] = 0b0. Storleken är 2^{20} byte vilket ger n = 19 som ger bitarna [5:1] = 0b10011. Denna regions konfigurationen skall gälla vilket ger bit [0] = 0b1. Detta ger CP15:c6:c1 = 0b100111 = 0x27.
- Region 0: Startadressen är 0 vilket ger bitarna [31:12] = 0b0. Storleken är 2^{30} byte vilket ger n = 29 som ger bitarna [5:1] = 0b11101. Denna regions konfiguration skall gälla vilket ger bit [0] = 0b1. Detta ger CP15:c6:c0 = 0b111011 = 0x3B.

5.3.1.2.2 Konfiguration av rättigheter

Konfigurationen av rättigheterna för respektive region görs i CP15:c5:c0, gäller instruktioner, och CP15:c5:c1, gäller data. I detta fall skall samma rättigheter gälla för instruktioner och data. CP15:c5:cx lägsta 16 bitar delas upp i 2-bitarsfält för respektive region, där region 0 har bitarna [1:0], region 1 bitarna [2:1] och så vidare. Rättigheterna ställs in enligt Kod-kolumnen i tabell 6.1.

Regionerna 1 och 0 har följande respektive bitmönster: 01 och 11. Detta ger CP15:c5:cx = 0b111 = 0x3. Slutligen sätts bit [0] = 0b1 i CP15:c1:c0 för att aktivera MPU:n.

5.4 Exekverande gästsystem sätter sin timer

Detta avsnitt beskriver hur ett gästsystem sätter sin timer. Beskrivningen utgår från den punkt då gästoperativsystemet anropar det hypercall som kallas *setTimer*. Alla hypercalls är definierade som en del av paravirtualiseringen.

Principen för att sätta ett gästsystems timer är enligt följande:

1. Gästoperativsystemet anropar *setTimer(int newTimerValue)*. Anropet lagrar önskat timervärde på en specifik minnesplats och genererar sedan ett mjukvaruavbrott.
2. Mjukvaruavbrottet gör att hypervisorn tar över exekveringen. Hypervisorn pausar timern och därefter placeras sedan gästsystemets nya timervärde i ett register. Sedan kan hypervisorn anropa timerkomponentens funktion *setCurrentVMsTimer(int newTimerValue)*.
3. Sedan startar hypervisorn timern och returnerar kontrollen till gästoperativsystemet.
4. Gästoperativsystemet återupptar exekveringen efter sitt *setTimer*-anrop.

En specifik algoritm som implementeras i assembler och C går till enligt följande (Appendix A.2 listar vad som finns på respektive minnesadresser):

1. Gästoperativsystemet anropar den paravirtualiserade funktionen *setTimerValue(int newTimerValue)*. Funktionen *setTimerValue* i skriven i C och utför följande steg:
    1. En pekare deklareras och sätts till 0x400. Detta är början på gästsystemets minnesregion. Denna minnescell används av gästsystemen för att ange deras nya timervärden.
    2. Minnescellen den deklarerade pekaren ovan pekar på sätts till parametervärdet

newTimerValue.

3. Sedan används en assemblerinstruktion inbäddad i C-språket. Detta är en SWI-instruktion (Software Interrupt) som orsakar ett mjukvaruavbrott. Detta gör att hypervisorn tar över exekveringen.

2. Efter att SWI-instrukionen exekverats sker följande i processorn:
    1. Processorn lagrar adressen av nästa instruktion (den efter SWI-instruktionen med adress PC – 4, vid mjukvaruavbrott subtraherar processorn PC med 4 automatiskt) i supervisor-lägets LR och CPSR i supervisor-lägets SPSR. Sedan modifieras CPSR så att dess innehåll representerar att processorn är i supervisor-läget.
    2. Programräknaren sätts till 0x8 där en instruktion finns som i sin tur sätter programräknaren till den adress som rutinen swi finns på.
    3. Programmet befinner sig nu i rutinen swi. Denna rutin tillhör hypervisorn och finns därför lagrad i region 1.
3. Alla register sparas undan på supervisor-lägets stack (r0-r12 och lr).
4. Fysiska timern pausas.
5. Ett funktionsanrop till *setCurrentVMsTimerValue(int newTimerValue)* utförs. Parameterregistret sätts till värdet som finns på adressen 0x10000, där gästoperativsystemets nya timervärde finns lagrat. Detta värde bör dock hypervisorn kontrollera så det inte orsakar overflow i timern. Detta steg förbises dock i denna fallstudie.
6. Fysiska timern startas**.**
7. Registerna återställs.
8. Returnering sker till gästoperativsystemets hypercall *setTimerValue*. Gästoperativsystemet kan sedan återuppta sin exekvering.

Koden för dessa åtta steg finns i appendix A.13.

5.5 Timeravbrott

Denna del beskriver hur ett timeravbrott behandlas av hypervisorn, och hur det levereras till aktuellt gästsystem. Det antas i denna fallstudie att endast timeravbrott kan ske, det vill säga inga andra I/O-enheter kan generera avbrott. Detta antagande betyder att: Nestlade avbrott inte kan uppstå (dock kan gästsystemen få nestlade avbrott); det behövs inte identifieras var avbrottet kommer ifrån; samt kontroll av exekveringsläge. Det sistnämnda kan annars resultera i att exekveringen lämnas över till gästsystemet i privilegierat läge (detta uppmärksammas läsaren på i avsnitt 5.5.1.2). Vidare antas även att processorn hamnar i IRQ-läget när ett timeravbrott uppstår.

För att förstå hur hela avbrottsprocessen går till, och för att illustrera skillnaden jämfört med avbrott utan hypervisor, underlättar det att förstå hur ett vanligt avbrott behandlas. Följande sekvens visar vilka steg ett avbrott går igenom i fallet utan hypervisor:

1. Vanlig exekvering utförs när ett avbrott sker.
2. Processorn sätter exekveringsläget till IRQ, lagrar PC i LR och CPSR i SPSR. Sedan sätts PC till 0x18 då det var ett IRQ-avbrott som orsakades.
3. I minnesplatsen på adress 0x18 finns en instruktion som gör att processorn går till en specifik rutin som behandlar avbrottet.
4. Avbrottsrutinen lagrar undan alla register på sin stack för att sedan behandla avbrottet.

5. Efter avbrottet behandlats återställer avbrottsrutinen registerna från stacken och lagrar värdet LR – 4 i programräknaren. Sedan återupptas den vanliga exekveringen från den punkt där den avslutades då avbrottet uppstod.

Det är detta som skall simuleras av hypervisorn. En överblick av principen för en sådan simulering går till enligt följande (implementationen, beskriven i avsnitt 5.5.1 skiljer sig dock):

1. Gästsystemet exekverar och ett avbrott sker.
2. Hypervisorn tar över kontrollen och ändrar gästsystemets programräknare till gästoperativsystemets avbrottsrutin.
3. Sedan lämnas kontrollen över till gästsystemet.
4. Gästoperativsystemet exekverar sin avbrottsrutin.
5. Sedan är det upp till gästsystemet att återta den tidigare vanliga exekveringen.

Detta gör att gästsystemet beter sig likadant som i fallet utan hypervisor. Den avbrottsrutin som gästoperativsystemet har måste dock ändras, som en del av paravirtualiseringen, för att kunna fungera i fallet med hypervisor. Ändringen gäller att avbrottsrutinen inte behöver lagra undan registerna då hypervisorn redan gjort det. Om ett gästsystem kan få flera olika typer av avbrott kan detta implementeras genom att hypervisorn har en pekare till respektive avbrottsrutin.

5.5.1 Algoritm för timeravbrott till aktuellt gästsystem

Hur den fiktiva hypervisorn kan hantera timeravbrott beskrivs i detta avsnitt. Det är denna del som bör optimeras för att ge snabba leveranser av realtidsavbrott. Algoritmen ser ut enligt följande (de kursiverade namnen är rutiner i implementationen):

1. Gästsystemet exekverar och ett avbrott sker.
2. Processorn lagrar CPSR_USER och PC i SPSR_IRQ respektive LR_IRQ. Sedan modifieras CPSR sådant att dess innehåll speglar att processorn är i IRQ-läget.
3. Programräknaren sätts till 0x18 där en instruktion finns som i sin tur sätter programräknaren till avbrottsrutinen *tint*.
4. Programmet befinner sig nu i rutinen *tint* som finns lagrad i hypervisorns minne. Registerna r0 till och med r12 samt LR_IRQ skall lagras undan för att möjliggöra registerna för användning. LR_IRQ – 4 är adressen på den instruktion gästsystemet skall exekvera efter sin avbrottsrutin. Registerna lagras undan på user-lägets stack enligt följande:
    1. r0: ursprungliga värde lagras på IRQ-lägets stack då det behövs i nästkommande steg.
    2. Processorns exekveringsläge ändras till system för att få tillgång till user-lägets stack där registerna skall lagras (system- och user-läget delar samma register vilket ger åtkomst till user-lägets stack-pekar-register). Således återställer gästsystemets avbrottsrutin registerna via sin egen user-stack (gästsystemet måste använda user-stacken då det exekverar i user-läget).
    3. Registerna r1-r12 och LR_IRQ lagras på user-lägets stack. Endast r0 återstår att lagras undan.
    4. Processorns exekveringsläge ändras till IRQ för att få tillgång till r0.
    5. r0:s ursprungliga värde tas bort från IRQ-lägets stack och lagras i just r0.
    6. Processorns exekveringsläge ändras till system för att få tillgång till user-lägets stack.

7. r0 lagras på user-lägets stack.
8. Processorns exekveringsläge ändras till IRQ. Notera att SPSR_IRQ innehåller gästsystemets CPSR_USER. CPSR_USER återställs när kontrollen lämnas över till gästoperativsystemets avbrottsrutin.

Nu är registerna fria för användning.

5. Anrop till *updateTimerComponent* utförs. Detta funktionsanrop uppdaterar timerkomponentens variabler och returnerar en kod för vem/vilka avbrottet är till. Om hypervisorn använder sin timer uppdateras fysiska timerns värde och därefter startas.
6. Beroende på returvärdet från *updateTimerComponent* görs följande:
   1. Avbrottet är till hypervisorn och därför sätts programräknaren till *interruptForHypervisor*.
   2. Avbrottet är till aktuellt gästsystem. Då lämnas kontrollen över till gästoperativsystemets avbrottsrutin för timeravbrott.
   3. Avbrottet är till båda. Då hypervisorn har prioritet i systemet sätts programräknaren till *interruptForHypervisor*. Det är upp till denna rutin att kontrollera om även gästsystemet skall få avbrott.

5.5.1.1 Fall 1 och 3: Avbrott för hypervisorn

7. Programräknaren sätts till hypervisorns avbrottsrutin. Denna rutin kan göra vad hypervisorn vill. De avslutande stegen skulle kunna vara steg 8 till 10.
8. Önskat timervärde sätts, timern uppdateras och startas.
9. Aktuell gäst byts ut och en ny byts in. Enligt metoden för att spara undan registerna så resulterar detta i att när en gäst byts in igen, finns alla dennes register redan på dess stack.
10. Antingen återupptas den vanliga exekveringen innan avbrottet uppstod. Eller i eventuellt fall fortsätter exekveringen av den nya aktuella gästens exekvering där den avslutade innan denne byttes ut.

5.5.1.2 Fall 2: Avbrott för gästsystemet

7. I annat fall är det endast aktuellt gästsystem som skall behandla avbrottet. Således skall exekveringen lämnas över till gästoperativsystemets avbrottsrutin. Exekveringen lämnas över enligt följande.

   Gästoperativsystemets avbrottsrutin för timeravbrott antas finnas på adress 0x401. Med hjälp av en instruktion sätts 0x401 i PC samtidigt som SPSR_IRQ sätts i CPSR. Det är i detta fall SPSR_IRQ måste kontrolleras så det inte innebär att gästsystemet exekverar i privilegierat läge. Detta fall tas ej hänsyn till då hypervisorn måste agera i de fall en överlämning av exekveringen resulterar i att gästsystemet exekverar i privilegierat läge.
8. Nu har gästoperativsystemet kontrollen och befinner sig i sin avbrottsrutin. Beroende på vad gästoperativsystemet önskas göra kan det antingen börja med att återställa valda registervärden för att analysera dessa eller utföra någon specifik uppgift för att slutligen återställa registerna.

   Sist återställs user-lägets stackpekare för att sedan ladda programräknaren med nästa instruktion i den vanliga exekveringen. Därefter återupptar gästsystemet sin tidigare exekvering innan avbrottet uppstod.

Kod för detta fall finns i appendix A.14.

5.5.1.3 Övriga fall

Det finns ytterligare fall att ta hänsyn till:

- Gästoperativsystemet vill sätta timern i sitt timeravbrott: Detta fall behandlas på samma sätt som om gästsystemet hade velat sätta timern under dess vanliga exekvering.
- Gästsystemet får ett timeravbrott under tiden det behandlar ett annat timeravbrott (detta nya timeravbrott kan uppstå genom fallet i punkten ovan): Hypervisorn ger gästsystemen möjlighet att återställa alla deras register och återuppta tidigare exekvering. Det är därför upp till gästsystemet att bearbeta nästlade avbrott.

6 Resultat

I avsnitt 3.3 listades följande krav och mål:

1. Säkerhetskrav: Inget gästsystem skall kunna påverka timern sådant att hypervisorn eller något annat gästsystem påverkas.
2. Hypervisorns timer skall räkna ned reell tid.
3. Varje gästsystems timer skall räkna ned dess motsvarande exekveringstid.
4. Samtliga operationer på timerkomponenten skall fullbordas i konstant tid enligt ordonotationen.
5. Minnesutrymmet timerkomponenten behöver får växa som snabbast linjärt med antalet gästsystem.
6. Timerkomponenten skall inte exekvera fler än 50 instruktioner från det att fysiska timern genererar en avbrottssignal tills dess att gästoperativsystemet börjat exekvera sin avbrottsrutin.
7. Hypervisorn skall inte exekvera fler än 50 instruktioner under samma exekveringssekvens som under punkt 6.

Hur dessa krav och mål uppnåddes beskrivs i efterföljande avsnitt. Hur kraven/målen i punkterna 2, 3, 4, och 5 uppnåddes beskrivs inte. Krav 2 och 3 beskrivs i avsnitt 4.2.2; krav 4 och 5 kan läsaren verifiera i appendix A.

6.1 Säkerhetskrav

I avsnitt 4.1.1.5 drogs slutsatsen att alla identifierade säkerhetshot skall motverkas av hypervisors då det är hypervisorn som skall hantera och skydda all hårdvara. Angående timerkomponentens korrekthet får resonemangen i avsnitt 5.2.2 samt koden i appendix A anses vara tillräckliga. Angående dimerkomponentens gränssnitt är det tre värden som bör kontrolleras: hypervisorns timervärde, gästsystemens timervärden, och den nya gästens ID vid byte av gäst. Det sistnämnda värdet är upp till hypervisorn att ange korrekt då hypervisorn har kontrollen över gästsystemen. Däremot bör timerkomponenten returnera felkoder om nya timervärden är utanför giltigt område: negativa eller genererar overflow. Kontroll av dessa värden gjordes inte då eventuella overflow beror på den använda timerns funktionalitet, och negativa värden kontrolleras enkelt i samband med overflow.

6.2 Tidsförskjutning av realtidsavbrott

I appendix A.15 finns assemblerkod som exekveras från det att timern genererar en avbrottssignal fram tills dess att gästoperativsystemets avbrottsrutin tar över exekveringen. Denna kod är omskriven från koden i appendix A.14 för att kunna identifiera vilka och antalet instruktioner som exekveras. Detta ger sedan en uppskattning av hur mycket ett realtidsavbrott förskjuts i tiden. Dock tas inte hänsyn till den specifika timerkommunikationen då ingen timer använts i denna fallstudie.

I tabell 6.1 visas antalet instruktioner och klockcykler hypervisorn exekverar som förskjuter realtidsavbrottet. Då hypervisorn lagrar undan fjorton register (och subtraherar stackpekaren) åt gästsystemet sparar gästsystemet tid. Således kan det påstås att hypervisorn bidrar med arton (33 – 15 = 18) instruktioner och 28 klockcykler (43 – 15 = 28, de kvarvarande instruktionerna tar en klockcykel).

Instruktion	Antal	Antal klockcykler	Totalt antal klockcykler
ADD	1	1	1
AND	2	1	2
LDR	1	1	1
MRS	4	1	4
MSR	4	3 (1)	12 (4)
ORR	2	1	2
STMIA*	12	12	12
STR	3	1	3
SUB	3	1	3
MOVS	1	3	3
	33		43 (35)

Tabell 6.1: Instruktioner som hypervisorn ger upphov till. Alla instruktioner förutom MOVS används för att lagra gästsystemets register på gästsystemets stack. MSR kan resultera i tre klockcykler beroende på om instruktionen ändrar kontrollflaggor i CPSR. STMIA är en instruktion som lagrar 12 register i minnet. Värdena för klockcyklerna är hämtade från [6] och gäller ARM9. Observera att hypervisorn lagrar undan registerna på gästsystemets stack, således behöver gästsystemet inte göra detta.

När ett gästsystem får ett avbrott kan timerkomponenten vara i ett av två tillstånd: både hypervisorn och gästsystemet använder sina timers, eller endast gästsystemet använder sin timer. I det förstnämnda fallet visar tabell 6.2 antal instruktioner (17) och klockcykler (19) som förskjuter realtidsavbrottet. I tabell 6.3 visas motsvarande antal instruktioner (16) och klockcykler (20) för fallet då endast gästsystemet använder sin timer. Maximala totala antalet instruktioner och klockcykler som både hypervisorn och timerkomponenten exekverar tillsammans är således 35 (= 18 + 17) respektive 48 (= 28 + 20). Målet var 50 + 50 instruktioner för timerkomponenten och hypervisorn, vilket är klart under 35. I det fall då både hypervisorn och gästsystemet använder sina timers måste timern både sättas och startas, medan i andra fallet måste endast timern sättas.

Instruktion	Antal	Antal klockcykler	Totalt antal klockcykler
ADD	1	1	1
BEQ	1	3	3
CMP	1	1	1
LDR	4	1	4
LSL	1	1	1
MOV	5	1	5
STR	3	1	3
SUB	1	1	1
	17		19

Tabell 6.2: Fallet då både hypervisorn och aktuell gäst använder sina respektive timers. Tabellen visar vilka instruktioner timerkomponenten ger upphov till vid leverans av realtidsavbrott. Observera att instruktioner för att sätta timern värde och starta den ej inkluderats.

Instruktion	Antal	Antal klockcykler	Totalt antal klockcykler
ADD	1	1	1
BEQ	2	3	6
CMP	2	1	2
LDR	3	1	3
LSL	1	1	1
MOV	5	1	5
STR	2	1	2
	16		20

Tabell 6.3: Fallet då endast gästsystemet använder sin timer. Endast timerkomponentens bidrag visas. Observera att kommunikation för att sätta timern bortses.

I och med att endast ett värde skall skickas till timern och sedan eventuellt starta timern via MMIO bör det räcka med ett fåtal (STR-) instruktioner för att göra detta. Ett rimligt antagande är att maximalt två instruktioner används för att skicka värdet och en för att starta timern. Således behövs maximalt cirka 40 instruktioner exekveras, och cirka 55 klockcykler (i ARM9-arkitekturen) passera innan gästsystemet kan behandla sitt avbrott. Målet var 100 instruktioner.

ARM926EJ-S från Texas Instruments har en klockfrekvens på 300 MHz. Med en sådan klockfrekvens tar 55 klockcykler 183,3 ns, och för en processor på 10 MHz, som diskuterades i avsnitt 3.2.3.1, tar det 5,5 µs. En sista observation är att i ARM-processorerna kan vissa minnesceller ”låsas” fast i cacheminnet. Därmed kan snabb data- och instruktionsåtkomst garanteras, plus att inga oförutsägbara tidsfördröjningar kan ske som i fallet vid cachemissar.

7 Diskussion

7.1 Design

7.1.1 Icke-parallell timerkomponent

Den timerkomponent som designades har endast förmågan att hålla reda på ett gästsystems exekvering i taget. Det vill säga gästsystemen erbjuds endast en timer som de får dela på. Exempelvis i fallet med servervirtualisering är det förmodligen önskvärt med en hypervisor som möjliggör parallell exekvering av ett flertal gästsystem. Då behövs ett godtyckligt antal timers samtidigt.

Detta kan lösas med en liknande metod som användes i detta projekt. Hypervisorn har en lista på de gästsystem som exekverar för tillfället. När fysiska timern skall sättas sätts det timervärde som är minst av gästsystemens och hypervisorns. Samtidigt registreras vems timervärde som sattes i fysiska timern. När sedan timern går ut subtraheras det minsta timervärdet (det som sattes i timern när denne startades) från hypervisorns och alla andra exekverande gästsystems timervärden. Sedan sätts det minsta av dessa uppdaterade timervärden i timern och därefter startas timern. Slutligen levereras avbrottet till motsvarande gästsystem.

7.1.2 Otillräcklig timerkomponent

Det specificerades i problemdefinitionen att timerkomponenten skulle fungera på ett sådant sätt att om ett gästsystem inte exekverar skall dess timer pausas. Anledningen till detta val var att det antogs att ett operativsystem sätter sin timer med avseende på dess exekveringstid. Bland annat när det skall bestämmas hur lång tid en applikation skall exekvera innan den byts ut. Denna sorts timervirtualisering passar bra för tillämpningar där datoranvändaren styr vilket gästoperativsystem som skall vara aktivt. Då beter sig gästoperativsystemet i princip identiskt med fallet utan hypervisor, sett ur ett timerperspektiv.

Däremot finns scenarion där detta designval av timerkomponent inte är lämpligt. Framför allt i fall då inbyggda system är involverade. Inbyggda system förekommer i bland annat säkerhetskritiska tillämpningar där realtidsavbrott är ett måste. Det vill säga att om ett gästsystem är utbytt och får ett timeravbrott byts det in igen och behandlar timeravbrottet. Exempelvis om ett antal operativsystem exekverar ovanpå en hypervisor i ett flygplan. I ett sådant fall kan exempelvis ett fördröjt timeravbrott resultera i att hastigheten inte mäts, vilket är ett måste för autopiloten. I fall med dataöverföringar används timern bland annat för att generera signaler till den andra kommunikationsenheten, som keep-alive och liknande. Om aktuellt gästoperativsystem byts ut kan kommunikationslänken brytas i onödan. Å andra sidan kan denna sorts timervirtualisering resultera i att mycket kontrollexekvering utförs, på grund av alla gästsystemsbyten, vilket försämrar effektiviteten.

Även denna form av design kan timerkomponenten anpassa sig efter. Genom att låta alla gästsystems timers räkna ned kontinuerligt. Ett krav hypervisors har på sig är att gästsystemen skall bete sig likadant vid egen hårdvara som vid en hypervisors sida. Denna sorts timervirtualisering kan skapa problem. Antag att ett gästoperativsystem byts ut efter att precis ha bytt in en applikation. När sedan detta gästoperativsystems timer går ut byts en ny applikation in oavsett om den förra applikationen exekverat.

Således kan slutsatsen dras att en timerkomponent som antingen endast räknar exekveringstid eller

endast reell tid är otillräcklig. Istället bör gästoperativsystemen erbjudas båda virtualiseringsmetoderna. Detta kräver dock paravirtualisering då gästoperativsystemen måste vara medvetna om de två olika metoderna för att räkna tid.

Utöver den sortens timeravbrott som detta projekt behandlat finns även andra sorters timers: systemklockor, synkroniseringsklockor, och timers som genererar periodiska avbrott. Således är en timerkomponent i en hypervisor komplex, och en timerkomponent bör kunna virtualisera alla dessa möjliga sorters timers.

7.1.3 Säkerhet

Som beskrevs i avsnitt 5.1.1 har en hypervisor ansvar för säkerheten. Därför behövde timerkomponenten i princip inte ta hänsyn till något säkerhetshot. Anledningen var att säkerhetshoten uppstod från datorns hårdvarukomponenter som hypervisorn skall skydda.

7.2 Implementation

7.2.1 Byten av exekveringslägen

En intressant aspekt vid leverans av avbrott till gästsystemen är hur lagringen av gästsystemens register går till. Genom att lagra dem på gästoperativsystemets stack är de lättåtkomliga och gästoperativsystemet slipper även utföra denna uppgift själv. Däremot krävdes det ett antal byten mellan processorns exekveringslägen. Dessa byten kräver ytterligare instruktioner vilket ökar tidsfördröjningen av realtidsavbrott, även om de är få till antal.

Ett snabbare sätt är att lagra registerna på en bestämd plats i gästsystemets minnesregion. Detta skulle ge upphov till 16 instruktioner istället för 33. Dock skulle systemet förlora dynamik om adresser låstes fast för varje sorts avbrott.

Enklast vore om processorn hade stöd för virtualisering genom att de privilegierade lägena har åtkomst till user-lägets stackpekare. Eller att de privilegierade lägena har en helt egen uppsättning av registerna r0 – r12.

7.2.2 Säkerhet

Ett problem som uppstår med denna implementation för leverans av avbrott är om processorn befinner sig i privilegierat läge när ett timeravbrott sker. Detta skulle kunna resultera i att gästsystemet exekverar i privilegierat läge när hypervisorn lämnar över exekveringen. Därför måste hypervisorn kontrollera i vilket exekveringsläge processorn var i innan avbrottet uppstod. Det vill säga om SPSR_IRQ innehåller ett CPSR som representerar ett privilegierat läge.

Hypervisorn bör även kontrollera timervärdena som gästsystemen önskar sätta för att undvika negativa värden eller overflow i fysiska timern.

7.2.3 Tidsfördröjning

Tidsfördröjningen av realtidsavbrott uppnådde målet om 100 instruktioner, cirka 40 instruktioner behövdes inklusive kommunikation med fysiska timern. Med en ARM9-processor på 300 MHz blev tidsfördröjningen under 200 ns. Detta ger en uppdateringsfrekvens på 5 MHz vilket får anses vara dugligt för de flesta tidskritiska tillämpningar.

Däremot som konstaterades i avsnitt 7.2.2 bör vissa säkerhetskontroller göras vilket påverkar

tidsförskjutningen av realtidsavbrott. I ett riktigt system behövs även andra kontroller göras: Exempelvis identifiera var avbrottet kom ifrån. Resultatet kommer dock fortfarande med största sannolikhet att vara under en µs.

7.3 Konceptet virtualisering

Konceptet virtualisering är mycket intressant och kan användas i många områden för att effektivisera hårdvaruutnyttjande, men även för att förenkla hårdvarusystem (istället för att ha två timers räcker det med en). En intressant tanke är att operativsystem virtualiserar många av datorns resurser för applikationerna som skall dela på dem. Med tanke på hur viktiga operativsystem är kan slutsatsen dras att virtualisering är ett mycket viktigt koncept för dagens datorsystem.

8 Referenser

[1] Jonathan G. Koomey. Growth in Data Center Electricity use 2005 to 2010. Analytics Press. 2011.

[2] GreenBiz [hemsida på Internet]. GreenBiz Group Inc. [uppdaterad 09-07-2010; Hämtad 13-04-2012]. Tillgänglig från: http://www.greenbiz.com/news/2010/07/09/most-servers-vastly-underutilized

[3] James E. Smith & Ravi Nair. Virtual Machines. San Francisco: Morgan Kaufmann 2005

[4] Andrew S. Tanenbaum. Modern Operating Systems. Prentice Hall 2007

[5] David A. Patterson & John L. Hennessy. Computer Organization and Design. San Francisco: Morgan Kaufmann 2004

[6] Andrew N. Sloss & Dominic Symes & Chris Wright. ARM System Developer's Guide. San Francisco: Morgan Kaufmann 2004

Appendix A

A.1 Timerkomponentens tillstånd

1. Hypervisorn har ett timervärde som är mindre än gästsystemets timervärde. Därför är det hypervisorns timervärde som sattes i fysiska timern när denne startades senast.
2. Hypervisorn har ett timervärde som är större än gästsystemets timervärde. Därför är det gästsystemets timervärde som sattes i fysiska timern när denne startades senast.
3. Endast Hypervisorn har ett timervärde. Fysiska timern påverkas på samma sätt som i fall 1.
4. Endast gästsystemet har ett timervärde. Fysiska timern påverkas på samma sätt som i fall 2.
5. Hypervisorn och aktuellt gästsystem har lika stora timervärden.
6. Varken Hypervisorn eller gästen har timervärde. Fysiska timern är därför i passivt läge.

A.2 Memory map

0x400 - Adress där nytt timervärde lagras när aktuellt gästsystem vill sätta sin timer.
0x401 - Adress där gästoperativsystemets avbrottsrutin för timeravbrott finns.
0x3FFFFFFF - Adress där gästsystemets stack börjar. Slut på region 0.
0x3FF - IRQ-lägets stack börjar på denna adress. slut på region 1.

Hypervisorns minnesregion: 0x0 – 0x3FF.
Gästsystemens minnesregion: 0x4FF – 0x3FFFFFFF.

Timerkomponentens variabler:
0x0 - currentState
0x4 - hypervisorsTimerValue
0x8 - currentVM
0xC – Arrayen VmsTimerValue början på 0xC.

A.3 StopHypervisorsTimer

```
void stopHypervisorsTimer()
{
	int *currentState = 0x0;
	int *hypervisorsTimerValue = 0x4;
	int *currentVM = 0x8;
	int *VMsTimerValue = 0xC + *currentVM; //Arrays base address added with the ID of the current VM.

	if (*currentState == 1) //HVs timer < Guests timer.
	{
		int hypervisorsTimerValue = 0x4;
		int elapsedTime = *hypervisorsTimerValue - getTimersValue();
		*hypervisorsTimerValue = 0;
		*VMsTimerValue = *VMsTimerValue - elapsedTime;
		*currentState = 4; //Only the guest is using a timer.
		setTimersValue(*VMsTimerValue);
	}
	else if (*currentState == 2 || *currentState == 5) //HVs timer > Guests timer or HVs timer = Guests timer
	{
		*hypervisorsTimerValue = 0;
		*currentState = 4; //Only the guest is using a timer.
	}
	else if (*currentState == 3) //Only HV using a timer.
	{
		*hypervisorsTimerValue = 0;
		*currentState = 6; //No one is using a timer.
		setTimersValue(0);
	}
}
```

A.4 stopCurrentVMsTimer

```
void stopCurrentVMsTimer()
{
	int *currentState = 0x0;
	int *hypervisorsTimerValue = 0x4;
	int *currentVM = 0x8;
	int *VMsTimerValue = 0xC + *currentVM; //Arrays base address added with the ID of the current VM.

	if (*currentState == 1 || *currentState == 5) //HVs timer < Guests timer or HVs timer = Guests timer.
	{
		*VMsTimerValue = 0;
		*currentState = 3; //Only the HV is using a timer.
	}
	else if (*currentState == 2) //HVs timer > Guests timer
	{
		int elapsedTime = *VMsTimerValue - getTimersValue();
		*hypervisorsTimerValue = *hypervisorsTimerValue - elapsedTime;
		*currentState = 3; //Only the HV is using a timer.
		setTimersValue(*hypervisorsTimerValue);
	}
	else if (*currentState == 4) //Only the guest is using a timer.
	{
		*VMsTimerValue = 0;
		*currentState = 6; //No one is using a timer.
		setTimersValue(0);
	}
}
```

A.5 setHypervisorsTimerValue

//Denna funktion skall inte användas för att nollställa timern, för det används stopXsTimer.

```
void setHypervisorsTimerValue(int newTimerValue)
{
	int *currentState = 0x0;
	int *hypervisorsTimerValue = 0x4;
	int *currentVM = 0x8;
	int *VMsTimerValue = 0xC + *currentVM; //Arrays base address added with the ID of the current VM.

	//Updates the HVs and the Guests respective timers.
	if (*currentState == 1)	//HVs timer < Guests timer.
	{
		int elapsedTime = *hypervisorsTimerValue - getTimersValue();
		*hypervisorsTimerValue = newTimerValue;
		*VMsTimerValue = *VMsTimerValue - elapsedTime;
	}
	else if (*currentState == 2 || *currentState == 4 || *currentState == 5) //HVs timer >= Guests timer.
	{
		*VMsTimerValue = getTimersValue();
		*hypervisorsTimerValue = newTimerValue;
	}
	else if (*currentState == 3 || *currentState == 6) //Only HV is using a timer.
	{
		*hypervisorsTimerValue = newTimerValue;
		*currentState = 3;
		setTimersValue(*hypervisorsTimerValue);
		return;
	}


	//Updates the current state and sets the physical timers value.
	if (*hypervisorsTimerValue < *VMsTimerValue)
	{
		*currentState = 1;
		setTimersValue(*hypervisorsTimerValue);
	}
	else if (*hypervisorsTimerValue = *VMsTimerValue)
	{
		*currentState = 5;
		setTimersValue(*hypervisorsTimerValue);
	}
	else if (*hypervisorsTimerValue > *VMsTimerValue)
	{
		*currentState = 2;
		setTimersValue(*VMsTimerValue);
	}
}
```

A.6 setCurrentVMsTimerValue

//Denna funktion skall inte användas för att nollställa timern, för det används stopXsTimer.

```
void setCurrentVMsTimerValue(int newTimerValue)
{
	int *currentState = 0x0;
	int *hypervisorsTimerValue = 0x4;
	int *currentVM = 0x8;
	int *VMsTimerValue = 0xC + *currentVM; //Arrays base address added with the ID of the current VM.

	//Updates the HVs and the Guests respective timers.
	if (*currentState == 1 || *currentState == 3 || *currentState == 5)		//HVs timer <= Guests timer.
	{
		*hypervisorsTimerValue = getTimersValue();
		*VMsTimerValue = newTimerValue;
	}
	else if (*currentState == 2) //HVs timer > Guests timer.
	{
		int timeElapsed = *VMsTimerValue - getTimersValue();
		*hypervisorsTimerValue = *hypervisorsTimerValue - timeElapsed;
		*VMsTimerValue = newTimerValue;
	}
	else if (*currentState == 4 || *currentState == 6) //Only the guest is using a timer.
	{
		*VMsTimerValue = newTimerValue;
		*currentState = 4;
		setTimer(*VMsTimerValue);
		return;
	}

	//Updates the current state and the physical timer.
	if (*hypervisorsTimerValue < *VMsTimerValue)
	{
		*currentState = 1;
		setTimersValue(*hypervisorsTimerValue);
	}
	else if (*hypervisorsTimerValue = *VMsTimerValue)
	{
		*currentState = 5;
		setTimersValue(*hypervisorsTimerValue);
	}
	else if (*hypervisorsTimerValue > *VMsTimerValue)
		*currentState = 2;
		setTimersValue(*VMsTimerValue);
	}
}
```

A.7 getHypervisorsTimerValue

```
int getHypervisorsTimerValue()
{
	int *currentState = 0x0;
	int *hypervisorsTimerValue = 0x4;
	int *currentVM = 0x8;
	int *VMsTimerValue = 0xC + *currentVM; //Arrays base address added with the ID of the current VM.

	if (*currentState == 1 || *currentState == 3 || *currentState == 5)
	{
		return getTimersValue();
	}
	else if (*currentState == 2)
	{
		int timeElapsed = *VMsTimerValue - getTimersValue();
		return *hypervisorsTimerValue - timeElapsed;
	}
	else if (*currentState == 4 || *currentState == 6)
	{
		return 0;
	}
}
```

A.8 getCurrentVMsTimerValue

```
int getCurrentVMsTimerValue()
{
	int *currentState = 0x0;
	int *hypervisorsTimerValue = 0x4;
	int *currentVM = 0x8;
	int *VMsTimerValue = 0xC + *currentVM; //Arrays base address added with the ID of the current VM.

	if (*currentState == 1)
	{
		int timeElapsed = *hypervisorsTimerValue - getTimersValue();
		return *VMsTimerValue - timeElapsed();
	}
	else if (*currentState == 2 || *currentState == 4 || *currentState == 5)
	{
		return getTimersValue();
	}
	else if (*currentState == 3 || *currentState == 6)
	{
		return 0;
	}
}
```

A.9 changeCurrentVM

```
void changeCurrentVM(int newVM)
{
	int *currentState = 0x0;
	int *hypervisorsTimerValue = 0x4;
	int *currentVM = 0x8;
	int *VMsTimerValue = 0xC + *currentVM; //Arrays base address added with the ID of the current VM.

	//Updates the hypervisors and the currentVMs values.
	if (*currentState == 1) {
		int timeElapsed = *hypervisorsTimerValue - getTimersValue();
		*hypervisorsTimerValue = getTimersValue();
		*VMsTimerValue = *VMsTimerValue - timeElapsed;
	} else if (*currentState == 2) {
		int timeElapsed = *VMsTimerValue - getTimersValue();
		*VMsTimerValue = getTimersValue();
		*hypervisorsTimerValue = *hypervisorsTimerValue - timeElapsed;
	} else if (*currentState == 3) {
		*hypervisorsTimerValue = getTimersValue();
	} else if (*currentState == 4) {
		*VMsTimerValue = getTimersValue();
	} else if (*currentState == 5) {
		*hypervisorsTimerValue = getTimersValue();
		*VMsTimerValue = *hypervisorsTimerValue;
	}

	//The code continues on next page. //Updates the new VMs values and the physical timer and the current state.
	*currentVM = newVM;
	VMsTimerValue = 0xC + *currentVM;

	if (*hypervisorsTimerValue == 0 && *VMsTimerValue == 0) {
		*currentState = 6;
		setTimersValue(0);
	} else if (*hypervisorsTimerValue != 0 && *VMsTimerValue == 0) {
		*currentState = 3;
		setTimersValue(*hypervisorsTimerValue);
	} else if (*hypervisorsTimerValue == 0 && *VMsTimerValue != 0) {
		*currentState = 4;
		setTimersValue(*VMsTimerValue);
	} else if (*hypervisorsTimerValue < *VmsTimerValue) {
		*currentState = 1;
		setTimersValue(*hypervisorsTimerValue);
	} else if (*hypervisorsTimerValue = *VmsTimerValue) {
		*currentState = 5;
		setTimersValue(*hypervisorsTimerValue);
	} else if (*hypervisorsTimerValue > *VmsTimerValue) {
		*currentState = 2;
		setTimersValue(*VMsTimerValue);
	}
}
```

A.10 updateTimerComponent

```
int updateTimerComponent()
{
	int *currentState = 0x0;
	int *hypervisorsTimerValue = 0x4;
	int *currentVM = 0x8;
	int *VMsTimerValue = 0xC + *currentVM;
	int interruptCase;

	if (*currentState == 2)		//Interrupt for guest. Affects delivery time to guest.
	{
		interruptCase = 2;
		*hypervisorsTimerValue = *hypervisorsTimerValue - *VMsTimerValue;
		*VMsTimerValue = 0;
		*currentState = 3;
		setTimersValue(*hypervisorsTimerValue);
		startTimer();
	}
	else if (*currentState == 4)		//Interrupt for guest. Affects delivery time to guest.
	{
		interruptCase = 2;
		*VMsTimerValue = 0;
		*currentState = 6;
		setTimersValue(0);
	}
	else if (*currentState == 1)		//Interrupt for HV.
	{
		interruptCase = 1;
		*VMsTimerValue = *VMsTimerValue - *hypervisorsTimerValue;
		*hypervisorsTimerValue = 0;
		*currentState = 4;
		setTimersValue(*VMsTimerValue);
	}
	else if (*currentState == 3)		//Interrupt for HV.
	{
		interruptCase = 1;
		*hypervisorsTimerValue = 0;
		*currentState = 6;
		setTimersValue(0);
	}
	else if (*currentState == 5)		//Interrupt for both.
	{
		interruptCase = 3;
		*hypervisorsTimerValue = 0;
		*VMsTimerValue = 0;
		*currentState = 6;
		setTimersValue(0);
	}

	return interruptCase;
}
```

A.11 initTimerComponent

```
void initTimer(int currentVM, int numberOfVMs)
{
	int *currentState = 0x0;
	int *hypervisorsTimerValue = 0x4;
	int *VMs;

	for (VMs = 8; VMs < numberOfVMs; VMs += 4)
		*VMs = 0;

	*currentState = 6;
	*hypervisorsTimerValue = 0;
	VMs = 0x8;
	VMs* = currentVM;
}
```

A.12 Vektortabellen

```
;Vector table. Only swi and tint are defined in this project.
B	start	;Branch to start routine
B	undef	;Branch to routine for undefined instruction exceptions
B	swi	;Branch to routine that handles software interrupts.
B	pab	;Branch to routine that handles prefetch abort exceptions
B	dab	;Branch to routine that handles data abort exceptions
		;This memory location is reserved
B	tint	;Branch to routine that handles IRQ interrupts, which is timer interrupts
B	pint	;Branch to routine that handles prioritized interrupts
```

A.13 Kod för att sätta gästsystemets timer

Koden i detta avsnitt används för att sätta gästsystemets timer.

A.13.1 Gästsystemets gränssnitt för att sätta timern

Denna kod tillhör gästoperativsystemet som anropar detta hypercall när det vill sätta sin timer. Funktionen lämnar överkontrollen till hypervisorn och förmedlar gästsystemets timervärde som skall sättas.

//This is the function that the paravirtualized guest operating system will execute when it wants to set a timer value. Observe that this function has nothing to do with the hypervisors timer component. The guest has no direct communication with the timer component, the hypervisor needs to protect that timer component.

```
void setTimerValue(int timerValue) {
        int *timerValueLocation = 0x400;        //The pointer to point to the first location of region 1.
        *timerValueLocation = timerValue;       //The pointer's memory location contains the
                                                //timer value.
        asm_volatile("swi");                    //GCC extended assembly, volatile means that the
                                                //compiler will not optimize this code and this
        return;                                 //instruction will be executed at its intended point.
}
```

A.13.2 Mjukvaruavbrott till hypervisorn

Denna kod exekverar hypervisorn.

```
swi:
        ;Stores the registers r0-r12 and svc_lr.
        SUB     svc_sp, svc_sp, #0x38     ;Makes room for 14 items on the stack. 14 * 4 = 56 = 0b111000 = 0x38
        STR     svc_lr, [svc_sp, #0x34]
        STR     r12, [svc_sp, #0x30]
        STR     r11, [svc_sp, #0x2C]
        STR     r10, [svc_sp, #0x28]
        STR     r9, [svc_sp, #0x24]
        STR     r8, [svc_sp, #0x20]
        STR     r7, [svc_sp, #0x1C]
        STR     r6, [svc_sp, #0x18]
        STR     r5, [svc_sp, #0x14]
        STR     r4, [svc_sp, #0x10]
        STR     r3, [svc_sp, #0x0C]
        STR     r2, [svc_sp, #0x08]
        STR     r1, [svc_sp, #0x04]
        STR     r0, [svc_sp, #0x00]

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;Stops the timer, timer specific code;;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;Sets the parameters for the function call setCurrentVMsTimerValue.
        MOV     r4, #0x101000             ;The memory location of the guest's new timer value.
        LDR     r0, [r4]                  ;r0 = The guest's new timer value.
        MOV     r4, #0x50                 ;The memory location of the hypervisor's timer value.
        LDR     r1, [r4]                  ;r1 = the hypervisor's timer value.
        MOV     r4, #0x51
        LDR     r2, [r4]                  ;r2 = current VM's ID.
        MOV     r4, #0x52
        LDR     r3, [4]                   ;r3 = base address of the array containing all VMs timer values.
        BL      setCurrentVMsTimerValue   ;Calls the c function for setting the current VM's timer.
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Starts the timer, timer specific code;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;Restores the registers.
LDR     r0, [svc_sp, #0x00]
LDR     r1, [svc_sp, #0x04]
LDR     r2, [svc_sp, #0x08]
LDR     r3, [svc_sp, #0x0C]
LDR     r4, [svc_sp, #0x10]
LDR     r5, [svc_sp, #0x14]
LDR     r6, [svc_sp, #0x18]
LDR     r7, [svc_sp, #0x1C]
LDR     r8, [svc_sp, #0x20]
LDR     r9, [svc_sp, #0x24]
LDR     r10, [svc_sp, #0x28]
LDR     r11, [svc_sp, #0x2C]
LDR     r12, [svc_sp, #0x30]
LDR     svc_lr, [svc_sp, #0x34]

;Returns to the return command in the setTimersValue function in the guest operating system. The instruction
;after the SWI instruction.
MOVS            pc, svc_lr          ;Sets PC to svc_lr and at the same time restores CPSR to SPSR
                                    ;of the supervisor mode.
```

A.14 tint

Denna kod exekveras när ett timeravbrott sker.

```
tint:
        ;Saves all registers.

        ;Saves r0 on IRQ's stack.
        SUB     irq_sp, irq_sp, #0x4        ;Makes room for one new item on the stack.
        STR     r0, [irq_sp, #0x0]          ;Puts r0_Value on the stack.
        ;-------------------------------------------------------------------------------------------------------

        ;Changes the processor's mode to system.
        MRS     r0, cpsr  ;r0 = CPSR.
        ORR     r0, r0, #0x1F               ;r0 = CPSR | 0x1F. Makes CPSR contain the bit value for system mode.
        MSR     CPSR, r0                    ;CPSR = r0. Sets the processor to system mode.
        ;-------------------------------------------------------------------------------------------------------

        ;Saves r1-r12 on user's stack. In total there are 14 items: r0-12 and LR. 14 * 4 = 56 = 0b111000 = 0x38.
        SUB     sp, sp, #0x38               ;Makes room for 14 new items on the user's stack.
        STR     lr, [sp, #0x34]
        STR     r12, [sp, #0x30]
        STR     r11, [sp, #0x2C]
        STR     r10, [sp, #0x28]
        STR     r9, [sp, #0x24]
        STR     r8, [sp, #0x20]
        STR     r7, [sp, #0x1C]
        STR     r6, [sp, #0x18]
        STR     r5, [sp, #0x14]
        STR     r4, [sp, #0x10]
        STR     r3, [sp, #0x0C]
        STR     r2, [sp, 0x08]
        STR     r1, [sp, #0x04]
        ;STR     r0, [sp, #0x00]            ;r0 will be saved later.
        ;-------------------------------------------------------------------------------------------------------

        ;Changes the processor mode to IRQ.
        MRS     r0, cpsr                    ;r0 = cpsr.
        AND     r0, r0, #0x12               ;r0 = CPSR & 0x12
        MSR     cpsr, r0                    ;CPSR = R0. Sets the processor's mode to IRQ.
        ;-------------------------------------------------------------------------------------------------------

        ;Pops r0 from IRQ's stack.
        LDR     r0, [irq_sp, #0x00]         ;r0 = r0_Value. Pops r0_Value off the stack.
        ADD     irq_sp, irq_sp, #0x8
        ;-------------------------------------------------------------------------------------------------------

        ;Changes the processor's mode to system.
        MRS     r1, cpsr                    ;r1 = CPSR.
        ORR     r1, r1, #0x1F               ;r1 = CPSR | 0x1F. Makes CPSR contain the bit value for system mode.
        MSR     CPSR, r1                    ;CPSR = r1. Sets the processor to system mode.
        ;-------------------------------------------------------------------------------------------------------

        ;Saves r0 on user's stack.
        STR     r0, [sp, #0x00]             ;Stores r0 on users stack.
        ;-------------------------------------------------------------------------------------------------------

        ;Changes the processor mode to IRQ.
        MRS     r0, cpsr                    ;r0 = cpsr.
```

```
        AND     r0, r0, #0x12           ;r0 = CPSR & 0x12
        MSR     cpsr, r0                ;CPSR = R0. Sets the processor's mode to IRQ.
        ;-----------------------------------------------------------------------------------------------------

        ;Now all registers are free for use.

        BL updateTimerComponent         ;Updates the timer component. r0 = interruptCase, the return value.

;-----------------------------------------------------------------------------------------------------

        CMP     r0, #0x1                ;Checks if r0 equals 1, this updates the zero flag i CPSR.
        BEQ     interruptForHypervisor  ;Branches to a routine that takes care of interrupts for the HV.
        CMP     r0, #0x3                ;Checks if r0 equals 3.
        BEQ     interruptForHypervisor  ;Branches to a routine that takes care of interrupts for the HV.

        ;The interrupt was for the guest.
        MOVS    pc, #0x401              ;Sets the program counter to the location of the guest operating system's
                                        ;timer interrupt routine. At the same time CPSR is set to SPSR_IRQ.
```

A.15 Optimerad tint för beräkning av antal instruktioner

```
tint:
        ;Saves all registers.

        ;Saves r0 on IRQ's stack.
        SUB     irq_sp, irq_sp, #0x4    ;Makes room for one new item on the stack.
        STR     r0, [irq_sp, #0x0]      ;Puts r0_Value on the irq-stack.
        ;-----------------------------------------------------------------------------------------------------

        ;Changes the processor's mode to system.
        MRS     r0, cpsr  ;r0 = CPSR.
        ORR     r0, r0, #0x1F           ;r0 = CPSR | 0x1F. Makes CPSR contain the bit value for system mode.
        MSR     CPSR, r0                ;CPSR = r0. Sets the processor to system mode.
        ;-----------------------------------------------------------------------------------------------------

        ;Saves r1-r12 on user's stack. In total there are 13 items: r1-12 and LR. 13 * 4 = 52 = 0b110100 = 0x34.
        SUB     sp, sp, #0x34           ;Makes room for 14 new items on the user's stack.
        STR     lr, [sp, #0x30]         ;Saves lr on the stack.
        STMIA   sp, {r1-r12}            ;Saves r1-r2 in ascending order.
        ;-----------------------------------------------------------------------------------------------------

        ;Changes the processor mode to IRQ.
        MRS     r0, cpsr                ;r0 = cpsr.
        AND     r0, r0, #0x12           ;r0 = CPSR & 0x12
        MSR     cpsr, r0                ;CPSR = R0. Sets the processor's mode to IRQ.
        ;-----------------------------------------------------------------------------------------------------

        ;Pops r0 from IRQ's stack.
        LDR     r0, [irq_sp, #0x00]     ;r0 = r0_Value. Pops r0_Value off the stack.
        ADD     irq_sp, irq_sp, #0x8
        ;-----------------------------------------------------------------------------------------------------

        ;Changes the processor's mode to system.
        MRS     r1, cpsr                ;r1 = CPSR.
        ORR     r1, r1, #0x1F           ;r1 = CPSR | 0x1F. Makes CPSR contain the bit value for system mode.
        MSR     CPSR, r1                ;CPSR = r1. Sets the processor to system mode.
        ;-----------------------------------------------------------------------------------------------------

        ;Saves r0 on user's stack.
```

```
        SUB     sp, sp, #0x4
        STR     r0, [sp, #0x00]             ;Stores r0 on users stack.
        ;---------------------------------------------------------------------------------------------------------

        ;Changes the processor mode to IRQ.
        MRS     r0, cpsr                    ;r0 = cpsr.
        AND     r0, r0, #0x12               ;r0 = CPSR & 0x12
        MSR     cpsr, r0                    ;CPSR = R0. Sets the processor's mode to IRQ.

        ;Now all registers are free for use.



;-------Here starts the timercomponents work-------------------------------------------------

        ;Checks which routine shall update the timer component, is in assembly for optimization.
        ;BL updateTimerComponent            ;Updates the timer component. r0 = interruptCase, the return value.

        ;Checks if interrupt is for the guest when both HV and the guest uses a timer.
        MOV     r0, #0x0;                   ;r0 = 0x0, address of currentState of timer component.
        LDR     r1, [r0]                    ;r1 = memory[r0], current state of timer component.
        CMP     r1, #0x2;                   ;Checks if timer component's state is 2.
        BEQ     currentState_is_2           ;Branches to currentState_is_2.

        ;Checks if interrupt is for the guest when only the guest uses a timer.
        MOV     r0, #0x0;                   ;r0 = 0x0, address of currentState of timer component.
        LDR     r1, [r0]                    ;r1 = memory[r0], current state of timer component.
        CMP     r1, #0x4;                   ;Checks if timer component's state is 4.
        BEQ     currentState_is_4           ;Branches to currentState_is_4.

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;Code that checks the other interrupt cases and acts accordingly.;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;---------------------------------------------------------------------------------------------------------
;Routines for updating the timer component when the interrupt is for the guest.

;Memory locations.
;currentState: 0x0
;hypervisorsTimerValue: 0x4
;currentVM: 0x8
;VMsTimerValue: arrayen börjar på adress 0xC

;The following code is translated into assembly.
;{
;       *hypervisorsTimerValue = *hypervisorsTimerValue - *VMsTimerValue;
;       *VMsTimerValue = 0;
;       *currentState = 3;
;       setTimersValue(*hypervisorsTimerValue);
;       startTimer();
;}

currentState_is_2:
        MOV     r1, #0x4                    ;r1 = memory location of hypervisors timer value.
        MOV     r2, #0x8                    ;r2 = memory location of the current guests ID.

        LDR     r3, [r1]                    ;r3 = hypervisors timer value.
        LDR     r4, [r2]                    ;r4 = current guests ID.
        LSL     r4, r4, #0x2                ;r4 = current guests ID * 4.
        ADD     r5, r4, #0xC                ;r5 = memory location of current guests timer value.
```

```
        LDR     r6, [r5]                ;r6 = current guests timer value.

        SUB     r0, r3, r6              ;r0 = hypervisors updated timer value.
        STR     r0, [r1]                ;Stores the hypervisors updated timer value.
        MOV     r6, #0x0                ;r6 = 0.
        STR     r6, [r5]                ;Clears current guests timer value.

        MOV     r7, #0x3                ;r7 = 0x3.
        STR     r7, [r6]                ;Sets the timer components state to 3.

        ;The instructions that the following comment implies increases the interrupt latency. This part
        ;is considered to be part of the hypervisor and nothing that the timer component can change.
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;Sets the physical timer with r0 as argument (HVsTimerValue) and then starts the physical timer;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;Instruction belonging to the hypervisor.
        MOVS    pc, #0x401              ;Lends the execution to the guest operating system, CPSR = SPSR_IRQ.

;---------------

;Memory locations.
;currentState: 0x0
;hypervisorsTimerValue: 0x4
;currentVM: 0x8
;VMsTimerValue: arrayen börjar på adress 0xC

;The following code is translated into assembly.
;{
;       *VMsTimerValue = 0;
;       *currentState = 6;
;       setTimersValue(0);
;}

currentState_is_4:
        MOV     r0, #0x8                ;r0 = memory location of the current guest ID.
        LDR     r1, [r0]                ;r1 = current guests ID.
        LSL     r1, r1, #0x2            ;r1 = current guests ID * 4.
        ADD     r2, r1, #0xC            ;r2 = memory location of current guests timer value.
        MOV     r0, #0x0                ;r0 = 0.
        STR     r0, [r2]                ;Saves the guests new timer value, 0.

        MOV     r4, #0x6                ;r4 = the new state of the timer component, currentState.
        STR     r4, [r3]                ;Saves the timer components new state.

        ;The instructions that the following comment implies increases the interrupt latency. This part
        ;is considered to be part of the hypervisor and nothing that the timer component can change.
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;Sets the physical timer with r0 as the timers functions argument, which is zero;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;Instruction belonging to the hypervisor.
        MOVS    pc, #0x401              ;Lends the execution to the guest operating system, CPSR = SPSR_IRQ.


;-------Here stops the timer components work when the interrupt is for the current guest-----------------------

;-------If the interrupt was not for guest it is to the hypervisor----------------------------------
        B       interruptForHypervisor  ;Branches to a routine that takes care of interrupts for the HV.
```