UPTEC F 01 074 Master´s degree project

SEP 2001

Time Accurate Simulation

MAGNUS NILSSON

1. Abstract

This report describes how an existing simulation technique for embedded systems can be extended to handle time accurate simulation. The simulation technique is based on separation of code into hardware dependent and hardware independent parts. The hardware independent part is compiled and run on a PC for simulation. The hardware dependent part is replaced with code that simulates the hardware on the PC. To perform time accurate simulation breakpoints are added to the source code. The breakpoints contain information about the execution time for the basic blocks of the code on the target system. By comparing the amount of time used on the target system between the nodes in the simulation a scheduler can synchronize their execution on the PC. The scheduler can also slow down the execution so it runs in the speed given by the times in the breakpoints. An implementation of a prototype system for Windows NT with the scheduler implemented as a DLL is described.

Innehållsförteckning

2. Bakgrund

2.1. Inledning

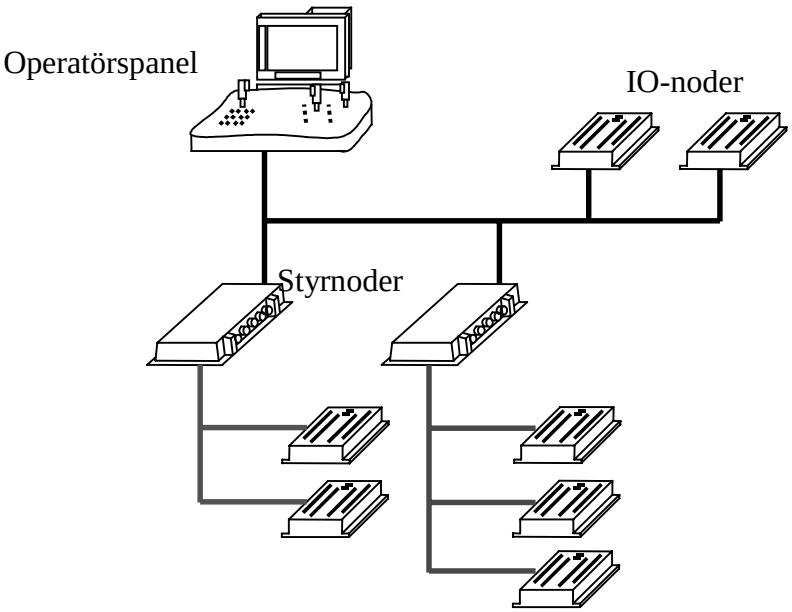
Avsikten med detta examensarbete är att utöka ett befintligt simuleringssystem så att det klarar tidsexakt simulering. I detta avsnitt studeras det system som ska utökas, ett simuleringssystem som utvecklats vid CC Systems[1].

2.2. Vad är simulering?

I det sammanhang som beskrivs av detta examensarbete är simulering en metod att efterlikna ett målsystem i en vanlig PC.

Ett målsystem består av en eller flera datorer, här kallade *noder*. Dessa noder har in- och utgångar för signaler till och från det system som skall styras, ofta ingår även något slags användargränssnitt (operatörspanel). Kommunikationen mellan noderna sker via ett nätverk.

Vid simulering körs alla eller några av dessa delar inuti en vanlig PC, nodernas in- och utsignaler skickas till en modell av det systemet skall styra.



Figur 1. Exempel på målsystem med flera noder.

Att göra simuleringen tidsexakt innebär att man kontrollerar nodernas exekveringshastighet då de simuleras i PC:n så att de även tidsmässigt beter sig som i målmiljön.

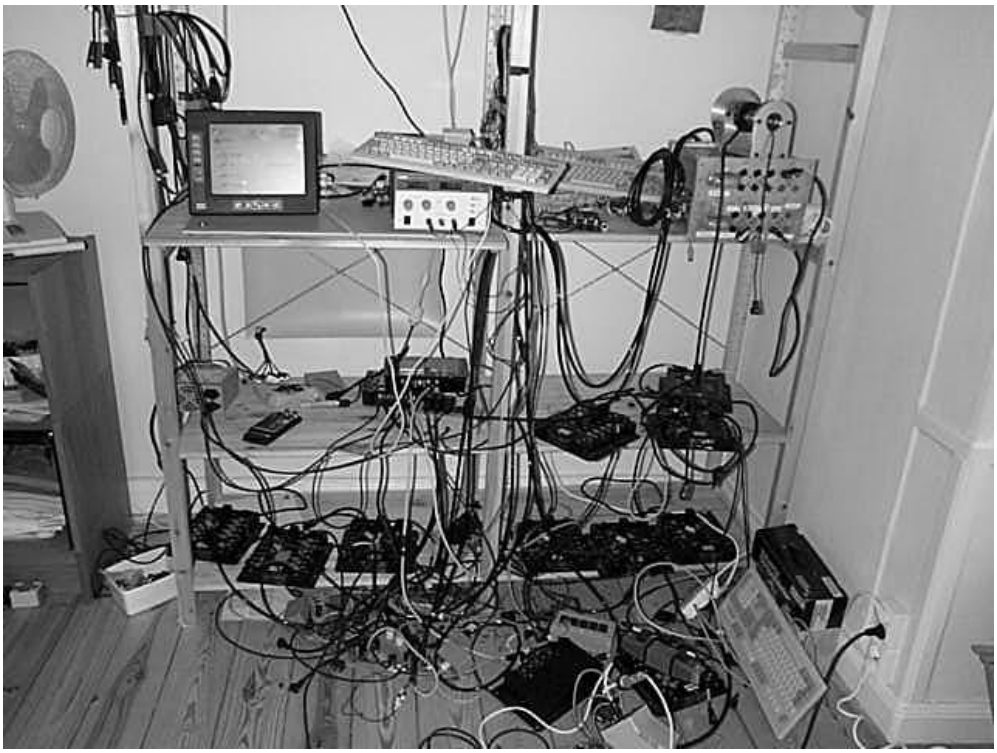
2.3. Varför vill man simulera?

CC Systems arbetar med utveckling av avancerade styrsystem. Styrsystemen blir allt mer komplexa och måste vara modulära och distribuerade. Om man utan simulering vill testa styrprogrammen under utvecklingsfasen måste de föras över till ett målsystem. Detta måste finnas tillgängligt, vara korrekt sammankopplat och strömförsett. Målsystemet saknar oftast goda debuggningsmöjligheter och

[1] Se mer om detta system i [2] och [5].

felsökning blir svårt. Då testning med hjälp av simulering kan utföras i varje programmerares PC på kontoret uppnås många fördelar:

- Alla utvecklare har alltid tillgång till en testmiljö och de praktiska problem som kan uppstå vid användning av en labbuppkoppling av målsystemhårdvara försvinner.
- Programvaran kan testas innan hårdvaran är färdigutvecklad eller innan hårdvaran finns tillgänglig.
- Då utveckling sker i Windowsmiljö kan de modernaste utvecklingsverktygen användas.
- Felsökning blir lättare då programflödet kan följas under simuleringen på ett sätt som inte är möjligt på målsystemhårdvaran.
- Omloppstiden för ändring, test och utvärdering blir betydligt kortare vilket leder till effektivare utveckling.
- Hårdvaran kan vara dyr och det är olämpligt att testa om det finns risk för allvarliga fel.

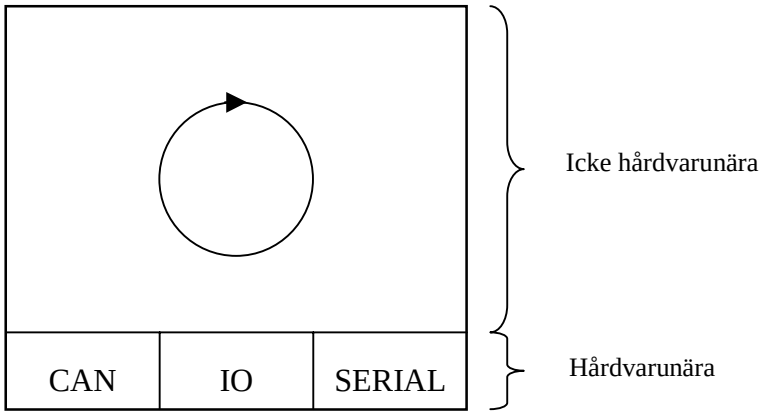


Figur 2. Så här kan en labbuppkoppling av ett målsystem se ut.

Simulering kan dock inte fullständigt ersätta tester på målsystemhårdvaran. Det finns fortfarande många viktiga skillnader mellan den simulerade miljön och målmiljön, t.ex. den hårdvarunära koden, operativsystem, kompilatorer/processorer och timing. Om man är medveten om dessa skillnader under utvecklingen kan de problem som skillnaderna skapar dock minimeras. Den tid som man sparar genom simulering kan sedan användas till bättre förberedd och mer konstruktiv testning på målsystemet.

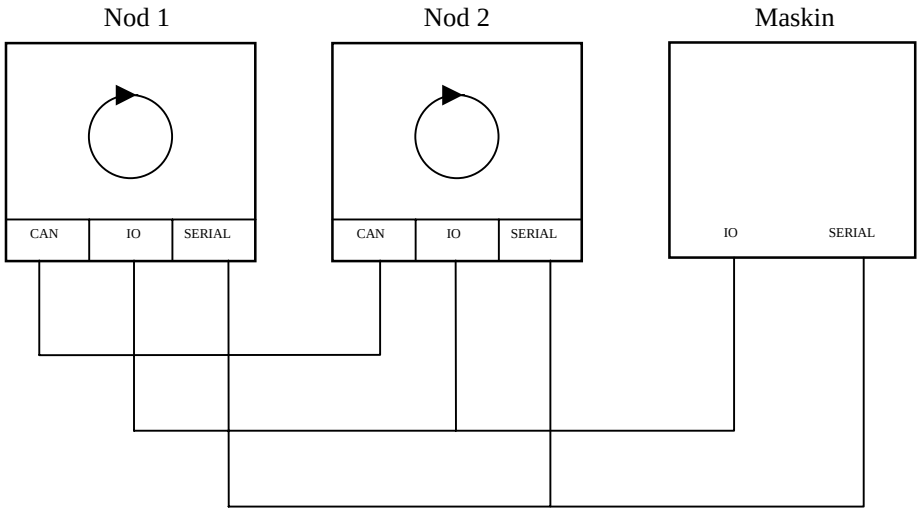
2.4. Simuleringens uppbyggnad

För att kunna utveckla ett system som både kan simuleras i en PC och köras på målsystemet måste man abstrahera bort hårdvaran med hjälp av en skiktad arkitektur. Koden i varje nod delas upp så att den hårdvarunära koden (åtkomst till CAN-nätverk, IO-portar osv) blir en separat del med ett enhetligt programmeringsgränssnitt till den icke hårdvarunära koden. Samma gränssnitt kan sedan implementeras i den simulerade miljön där hårdvarans funktioner simuleras. Den icke hårdvarunära koden blir på detta sätt densamma i den simulerade miljön och i målmiljön, vilket är det man vill uppnå.



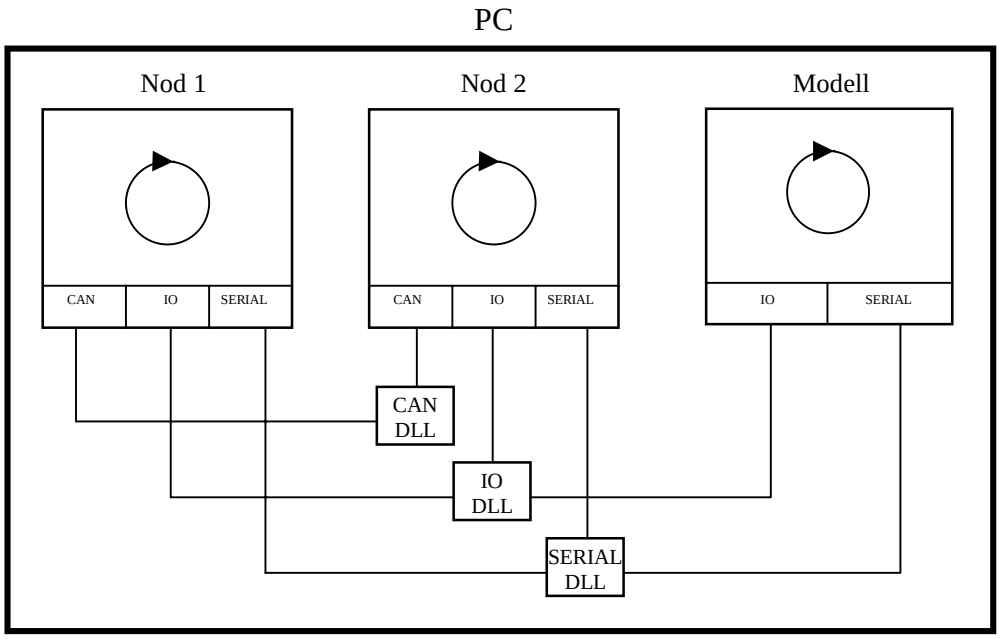
Figur 3. Uppdelningen av kod i en nod.

I målsystemet måste den hårdvarunära koden skrivas om för varje ny typ av hårdvara som används. Men eftersom samma typ av hårdvara ofta används i flera noder eller i olika projekt behöver detta inte kosta alltför mycket.



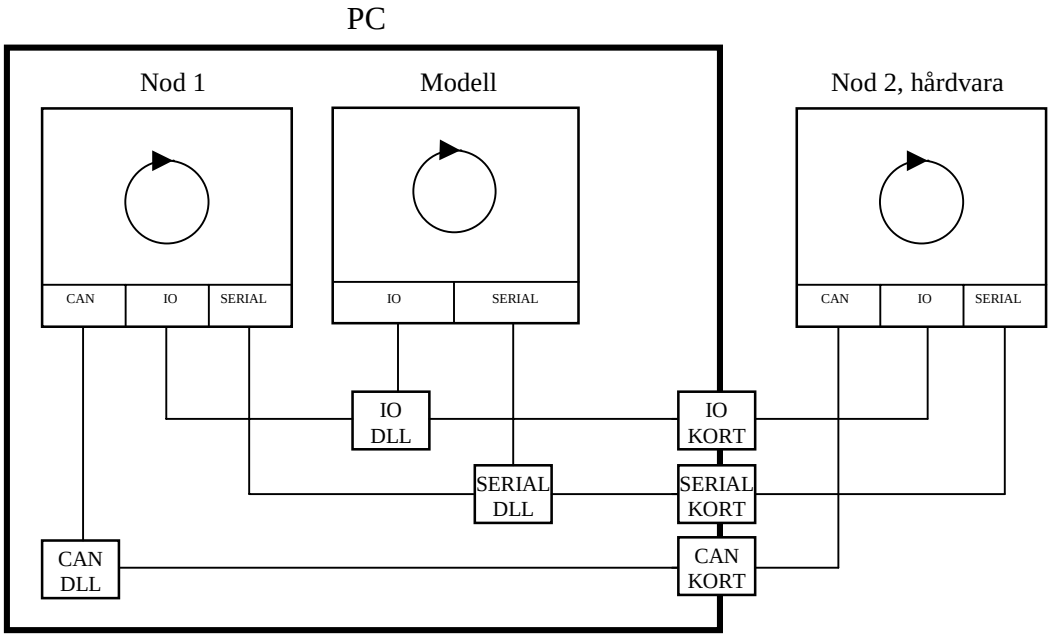
Figur 4. Exempel på målsystem. Två noder som kommunicerar via CAN samt en maskin som styrs via IO-signaler och en seriell buss.

I simuleringsmiljön har CC Systems implementerat simulerade gränssnitt för t.ex. CAN och IO, så att den icke hårdvarunära koden kan köras och funktionellt bete sig som i målsystemet. Flera noder körs som separata processer inom samma PC och de simulerade gränssnitten kan kopplas samman till virtuella nätverk så att ett komplett system kan ”kopplas ihop” i datorn.



Figur 5. Målsystemet i figur 4 simuleras här i en PC, maskinen beskrivs av en modell.

Genom att installera hårdvarukort i simuleringsdatorn kan de interna virtuella signalerna och bussarna kopplas samman med målsystemets signaler och bussar. En simulerad nod kan på detta sätt styra en verklig maskin, eller vice versa. Man kan på detta sätt t.ex. testa en färdig nod i hårdvara mot en modell av en maskin som ej finns tillgänglig.



Figur 6. Blandad simulering. En av noderna körs i målmiljön och resten av systemet simuleras.

3. Synkronisering

3.1. Inledning

När ett system simuleras körs varje nod som en separat process i simuleringsdatorn. Med den befintliga simuleringstekniken sköter Windows NT hur simuleringsdatorns processortid fördelas mellan processerna. Varje process kör så fort den kan och det sker ingen tidssynkronisering mellan processerna. Oavsett vilka skillnader i hastighet som finns mellan noderna i målsystemet körs de i simuleringen i samma hastighet. För att kunna synkronisera processerna så att de kör med rätt relativ hastighet måste kontrollen över deras exekvering till stor del föras över från operativsystemet till en egen applikation.

3.1.1. Varför vill man synkronisera processerna?

Målet med simuleringen är att så väl som möjligt efterlikna målsystemets beteende i en vanlig PC. Om processerna som simulerar noderna i ett system tillåts köra i ett tempo som godtyckligt sätts av PC:ns operativsystem och processorhastighet kommer de inte att bete sig som i målsystemet.

Antag att en process kör snabbare än en annan process trots att noderna de simulerar är lika snabba i målsystemet. Om de kommunicerar med varandra med någon form av meddelanden kan problem uppstå. Den snabba processen kommer att skicka meddelanden för tidigt (enligt en utomstående betraktare) eftersom den går för snabbt. Den långsamma processen kommer att få meddelanden vid fel tidpunkt, eftersom den snabba processen skickar dem för tidigt och den långsamma processen lyssnar för sent, kanske hinner den inte ens ta emot dem.

Även om nodernas program och kommunikationsprotokoll skrivs så att de klarar av förskjutningar av detta slag så kommer de testkörningar man gör i den simulerade miljön inte visa upp nodernas verkliga beteende. Meddelanden kan exempelvis komma i en annan ordning vid simulering jämfört med i målsystemet, detta kan skapa felaktigheter som inte upptäcks i simuleringen.

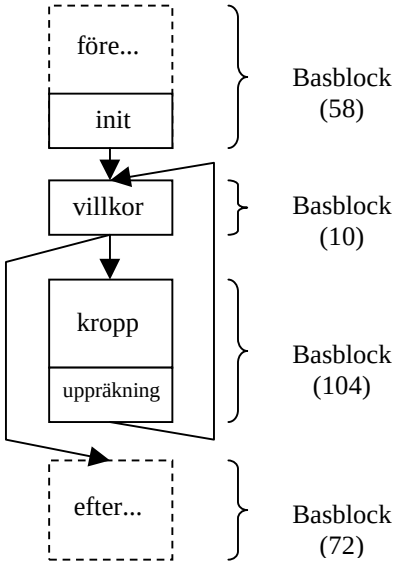
3.2. Målsystemets exekveringshastighet

Då simuleringen skall efterlikna målsystemet måste man veta nodernas exekveringshastighet. Det kan vara svårt att beräkna exekveringstider då koden skrivits i ett högnivåspråk, där man inte vet hur många instruktioner som egentligen utförs och koden kan gå många olika vägar beroende på villkorssatser. I detta arbete antar jag att koden delats upp i korta block med känd exekveringstid på målsystemet. Dessa exekveringstider kan man få från framtida kompilatorer (utveckling pågår) eller genom mätningar på målsystemet. Man kan även manuellt anta ”worst case”-tider för varje block.

3.2.1. Basblock

För att kunna ange en exekveringstid för ett block måste det vara ett block som kan exekveras utan hopp och ej innehåller flera olika möjliga exekveringsvägar. Ett block med dessa egenskaper kallas ett basblock[2]. Som exempel studerar vi en enkel for-loop.

```
före...
for (init; villkor; uppräkning)
{
   kropp;
}
efter...
```



Figur 7. Exempel som visar hur en for-loop delas upp i basblock.

For-loopen har först en initieringsdel, den blir ett basblock (om möjligt den sista delen av basblocket före for-loopen). Sedan har den ett villkor, detta blir också ett basblock. Om villkoret var sant körs for-loopens kropp, denna kan bestå av ett eller flera basblock, i detta exempel antar vi att kroppen ej innehåller villkorssatser och ej har flera möjliga exekveringsvägar. Detta medför att kroppen består av ett basblock. For-loopen har sedan en uppräkningssats, denna knyts om det är möjligt till slutet av kroppen så att de ingår i samma basblock.

3.2.2. Tider från kompilatorn

En kompilator kan skrivas så att den vid kompileringen beräknar det antal processorcykler som behövs för att exekvera varje basblock. Man kan sedan skapa en kopia av källkoden där slutet av varje basblock annoteras med beräknad exekveringstid i antal processorcykler. Denna annotering kan sedan omvandlas till brytpunkter som används för tidskontroll då koden kompileras för att köras i simulerad miljö.

```
före;
for ( init, cyclecount(58);
      villkor, cyclecount(10);
      uppräkning, cyclecount(104) )
{
   kropp;
}
efter;
cyclecount(72);
```

Figur 8. Koden från föregående figur efter annotering. Annoteringen för for-loopens kropp sitter efter uppräkningen eftersom kroppen och uppräkningen slås ihop till ett basblock.

[2] Se [4] Muchnick, Advanced Compiler Design and Implementation

De tider som kompilatorn kan beräkna anges i processorcykler vilka är beroende av målsystemets specifika klockfrekvens. I det simulerade systemet kan flera målsystem med olika klockfrekvenser ingå och tiden för deras basblock måste konverteras från processorcykler till någon global enhet. Att finna en representation för denna enhet i det simulerade systemet med hög noggrannhet kan medföra vissa svårigheter.

Det finns flera nackdelar och svårigheter med denna metod att finna exekveringstiden. Om kompilatorn använder aggressiv optimering kan det medföra att basblocken i den optimerade koden ej nödvändigtvis är de samma som i källkoden, det blir därmed omöjligt att annotera källkoden med basblockstider. Om processorn i målsystemet använder pipelining kan de tider som kompilatorn beräknat bli missvisande, exekveringen av ett nytt basblock kan påbörjas innan det föregående slutförts och överlappningen syns ej i annoteringen, eller så måste man approximera vilket leder till minskad precision.

3.3. Brytpunkter

För att kunna kontrollera exekveringen av varje process läggs en brytpunkt in i slutet av varje tidsbestämt block i koden. Vid simuleringen körs kodblocket i full fart på simuleringsdatorn, när brytpunkten nås sätts en räknare till nodens nya målsystemtid (adderar den kända exekveringstiden för blocket till räknaren). Genom att under simuleringen bevaka varje process räknare för målsystemtid kan processer som kommit före de andra stoppas så att alla noder har ungefär samma målsystemtid.

Brytpunkterna kan läggas in manuellt vid de punkter i koden mellan vilka man mätt målsystemets exekveringstid eller så kan de läggas in automatiskt med hjälp av information från kompilatorn (som beskrivits ovan). Brytpunkterna består antingen av ett funktionsanrop eller av en kort kodsnutt. Om man lägger in brytpunkter mellan varje basblock blir de väldigt många och mycket processortid kommer att användas av brytpunkternas kod, man får en stor overhead. Man måste därför göra den kod som körs vid varje brytpunkt så kort och snabb som möjligt. Alternativt kan man välja att inte göra allt vid varje brytpunkt utan vänta med att göra funktionsanropet tills brytpunkterna samlat på sig en viss mängd målsystemtid.

3.4. Schedulering

Då simuleringsdatorn antas vara ett enprocessorsystem måste alla noders kod köras på dess enda processor. Normalt sköter operativsystemet Windows NT scheduleringen, men för att få mer kontroll över den ordning i vilken trådarna körs och för att få bättre tidsupplösning med kortare exekveringstid för varje tråd har jag valt att delvis styra detta själv.

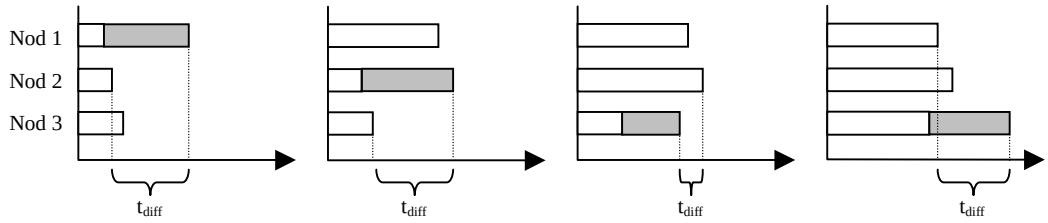
Jag har studerat olika metoder för att kontrollera hur processer och trådar exekveras och suspenderas i Windows NT. Under Windows NT fann jag två enkla metoder som påverkar operativsystemets egen schemaläggare så att den beter sig på önskvärt sätt. Den ena metoden baseras på att man ändrar processernas prioritet i operativsystemet under körning och därmed tvingar operativsystemets schedulerare att köra vissa trådar hellre än andra. Den andra metoden använder operativsystemets stöd för Mutual Exclusion med vars hjälp man kan hindra alla processer och trådar (som ingår i simuleringen) utom en från

att exekvera. I implementationen har jag använt den senare metoden då den kan ge total kontroll över vilken process som körs. Den prioritetsbaserade metoden kan inte användas för att helt stoppa exekveringen av en viss process och är därför olämplig.

3.4.1. Synkroniseringsmetod

För att hålla alla noder synkroniserade låter jag den process som har lägst målsystemtid köra, alla andra står stilla. Vid varje brytpunkt görs en kontroll av processernas målsystemtid och om en annan process då har lägst målsystemtid övertar den processorn. På detta sätt hålls målsystemtiden hos de olika processerna på samma nivå. Skillnaden i målsystemtid mellan två processer är lika med eller mindre än den största existerande kodblockstiden. Detta inses lätt eftersom den process som kör alltid är den med lägst målsystemtid och den kan därför aldrig komma längre före de andra processerna än den tid den får köra, en kodblockstid.

$$t_{diff} \leq \max(t_{block})$$



Figur 9. Det kodblock från den nod som har lägst målsystemtid får exekvera. Skillnaden i målsystemtid mellan två processer är lika med eller mindre än den största existerande kodblockstiden.

3.4.2. Noder med flera trådar och interrupt

Då en nod innehåller mer än en tråd uppstår ett nytt problem. Målsystem-tiden måste vara densamma för alla trådar som tillhör samma nod. Om en tråd exekverar förbrukar den processortid i noden och ”tar” tid från alla andra trådar i samma nod. Samma sak gäller för interrupt, då ett interrupt exekverar i en nod ökar målsystemtiden för alla trådar i noden eftersom interruptet kräver en viss mängd processortid. Genom att utse en huvudtråd i varje nod som hanterar målsystemtiden och genom att låta alla andra trådar och interrupt i den noden uppdatera denna tid då de arbetar så kan man få en enhetlig målsystemtid för noden.

Prioriteter

Då målet med detta arbete är att hålla flera noder synkroniserade tidsmässigt så har målsystemtiden störst inverkan vid valet av vilken tråd som skall få exekvera först. Om flera trådar har samma målsystemtid (vilket ofta händer om man har många trådar eller interrupt i en nod, de har ju då samma målsystemtid) bör den tråd som har högst prioritet köras. Prioriteten sätts av programmeraren. Om ett interrupt inkommer har det hög prioritet och får därmed köra först (om ingen annan ligger efter och har lägre målsystemtid). Om flera trådar har samma målsystemtid och samma prioritet får någon av de trådar som ej nyss kört och som först ställde sig i kö exekvera, detta system har jag valt för att undvika att någon process aldrig får köra (starvation). En tråd med hög prioritet kan dock ej

avbryta ett pågående block med lägre prioritet utan måste vänta till blocket är färdigt, detta kan medföra en del problem med interrupt vilket diskuteras mer senare. I ett målsystem som har noder med flera trådar styrs antagligen trådarna i noderna av ett RTOS. När en sådan nod simuleras bör trådschedulering och hantering av prioriteter anpassas så att de efterliknar beteendet hos målsystemets RTOS.

3.5. Verklig tid, blandsimulering

Tidigare har endast intern synkronisering mellan de simulerade noderna i simuleringsdatorn diskuterats. Det har inte funnits några krav på att simuleringen ska följa någon verklig tid. Då de simulerade noderna i simuleringsdatorn kommunicerar med verkliga noder i hårdvara utanför datorn måste simuleringen bromsas upp så att den kör i verklig tid. Detta kan åstadkommas genom att endast låta den process som har lägst målsystemtid köra om simuleringsdatorns systemklocka (relativt simuleringens start) har ett värde som är större än målsystemtiden. Annars får alla processer vänta.

På detta sätt kan man även få simuleringen att ske i valfri hastighet genom att multiplicera systemklockans värde med en konstant, om konstantens värde är mindre än ett körs simuleringen i slowmotion. Om man använder tidsrelaterade kommandon direkt från operativsystemet som t.ex. Sleep (för att suspendera en process en viss tid) i de program som synkroniseras bör man tänka på att dessa ej påverkas av att tidskonstanten ändras och det kan ge konstiga fel.

3.6. Problem, svårigheter

För att ovanstående resonemang skall vara giltigt krävs att simuleringsdatorn är mycket snabbare än de noder som simuleras. Utöver belastningen av processerna som simulerar noderna tillkommer overhead för kontroll av målsystemtider och processbyten. Då inbyggda system oftast är baserade på ganska enkla och billiga processorer och då dagens PC-maskiner är mycket snabba är detta oftast inte ett problem.

Synkroniseringsmetoden jag föreslagit ovan synkroniserar de processer som ingår i det simulerade systemet. Operativsystemet, Windows NT, kan dock när som helst avbryta de processer som kör simuleringen för att köra andra program, ta emot nätverksmeddelanden, behandla interrupt osv. Då simuleringen enbart består av processer i simuleringsdatorn är detta ej så farligt, alla simulerade processer fördröjs lika mycket och synkroniseringen kan bibehållas. Då simuleringen ska köras i verklig tid (t.ex. vid blandsimulering) kan dock problem uppstå. Ett avbrott från operativsystemet kan fördröja en tidskritisk process så att delar i systemet som inte fördröjts på samma sätt tappar synkroniseringen.

Då kommunikation mellan noderna sker måste man ta hänsyn till att ett svar på ett meddelande ej kan fås förrän den andra noden getts möjlighet att köra. Om en tråd tillhörande en nod ställer sig och väntar på ett meddelande från en annan nod kan meddelandet inte mottas förrän exekveringen av den första noden avbryts, den andra noden tillåts köra och den ursprungliga noden sedan körs igen. Om ett program skrivs oförsiktigt så att exekveringen ej kan avbrytas då programmet väntar på ett meddelande utifrån kan deadlock-situationer uppstå. Om varje loop i programmet innehåller minst en brytpunkt undviks detta problem.

4. Representation av tid

4.1. Inledning

Alla simulerade noder måste representera sin målsystemtid i en form så att jämförelse mellan nodernas tider i det simulerade systemet blir möjlig. Den datatyp som används måste kunna representera mycket korta tider med hög exakthet samtidigt som det är önskvärt att den kan lagra tider stora nog för flera timmars simulerade körningar. Typen bör helst vara någon form av heltal för att minimera beräkningstiden vid jämförelser och beräkningar.

4.2. Lämpliga enheter och datatyper

Att använda sekunder som enhet verkar uppenbart men det kan skapa problem. Om sekunder används måste man använda flyttal för att representera tiden. Då flyttal blir stora minskar deras precision. Dessutom tar beräkningar och jämförelser med flyttal längre tid för datorn att utföra.

4.2.1. Flyttal

Om man använder Visual C++ på en Intel x86 processor består datatypen float av 4 bytes, 8 bitars exponent och 23 bitars mantissa. Detta ger 6 till 7 signifikanta siffror. Om man vill kunna representera tid med en upplösning av ca 1/100 μs (1 μs $\Rightarrow$ 1 MHz) medför det att det största tal som kan anges med denna precision är mindre än 0,01 s. Detta är inte acceptabelt och datatypen float kan alltså inte användas.

Datatypen double består av 8 bytes, 11 bitars exponent och 52 bitars mantissa. Detta ger 15 till 16 signifikanta siffror. Om man vill kunna representera tid med en upplösning av ca 1/100 μs medför det att det största tal som kan anges med denna precision är mindre än 10^8 s ≈ 27777 timmar ≈ 3 år. Detta borde vara mer än tillräckligt för de flesta tänkbara simuleringar.

4.2.2. Heltal

Om man istället för flyttal använder heltal för att representera tiden får man bättre prestanda vid jämförelser och beräkningar. I Visual C++ består datatypen long av 4 bytes (32 bitar). Eftersom vi inte använder negativ tid behövs ingen teckenbit. Om man som tidigare vill kunna representera tid med en upplösning av 1/100 μs blir den största tid som kan anges 1/100 * 2^{32} μs ≈ 42,9 s.

Man kan även definiera egna heltalstyper med 8, 16, 32 eller 64 bitar. Med ett 64 bitars heltal (utan teckenbit) och med samma upplösning som tidigare blir den största möjliga tiden ungefär 5800 år.

4.3. Konvertering

De tider som hanteras i det simulerade systemet (systemtiden från operativsystemet och exekveringstiderna för de olika nodernas kodblock) kommer från flera olika källor.

Systemtiden från operativsystemet Windows NT kan fås i många olika former. Den som har högst upplösning och passar bäst i detta sammanhang är QueryPerformanceCounter. Med funktionen QueryPerformanceFrequency får man dess upplösning vilken är ca 1 μs.

Exekveringstiderna för nodernas kodblock kommer från mätningar, från antaganden eller från en kompilator. Då de kommer från mätningar eller antaganden är de antagligen angivna i ms eller µs, då de kommer från en kompilator är de angivna i klockcykler. De olika noderna i systemet kan vara baserade på olika typer av processorer med olika klockfrekvenser så antalet klockcykler måste anges tillsammans med respektive processors klockfrekvens för att informationen ska vara meningsfull.

4.3.1. Gemensam enhet

För att dessa tider ska vara jämförbara måste de konverteras till en gemensam enhet, var ska denna konvertering ske och hur ska den gemensamma enheten se ut? Konverteringen kan antingen ske direkt vid källan, i gränsnittet till den gemensamma delen av koden eller alldeles innan en jämförelse eller beräkning görs. För att undvika svårigheter med lagring av olika datatyper och för att minimera förvirring orsakad av flera parallella enheter har jag valt att försöka flytta konverteringen så nära källan som möjligt. Några exempel:

- I samband med ett anrop till systemklockan konverteras den returnerade tiden till den gemensamma enheten.
- Då kod annoterad med exekveringstider från en kompilator förses med brytpunkter konverteras klockcyklerna till den gemensamma enheten.

Detta ställer höga krav på den gemensamma enheten då den måste ha tillräckligt hög noggrannhet för att sköta lagring och hantering av alla tider i systemet.

Exekveringstiderna för en nods basblock som fås från en kompilator mäts i klockcykler. De klockfrekvenser som används i nodernas inbyggda datorer ligger oftast mellan 1-25 MHz. En klockcykel blir alltså mellan 40-1000 ns lång. Ofta används klockfrekvenser som ej ger jämna tider i nanosekunder, t.ex. 6,33 MHz ger 157,9778... ns, för att avrundningsfelen inte ska bli alltför stora då många klockcykler adderas bör upplösningen på den gemensamma enheten vara bättre än en nanosekund.

I avsnittet ovan visades att 32 bitars heltal ej klarar tillräckligt stora tal med hög noggrannhet, 64 bitars heltal har dock flera goda egenskaper för att lagra tid. I exemplet ovan användes upplösningen 1/100 µs vilket gav möjligheten att lagra onödigt stora tider. Om upplösningen sätts till 1 pikosekund (10^{-12} s) blir den största möjliga tiden ungefär 5100 timmar (≈ 210 dagar) vilket fortfarande är mer än nödvändigt.

Med 1 pikosekunds upplösning och 6,33 MHz klockfrekvens kan en klockcykels tid beskrivas med ett fel på $1.17 \cdot 10^{-13}$ s, vilket är ca 0,000074 % av en hel klockcykel.

5. Hårdvarunära kod och interrupt

5.1. Inledning

Vi vill gärna simulera interrupt då de ofta är källan till problem som kan vara svåra att förutse men de är tyvärr svåra att simulera då de är starkt knutna till nodernas hårdvara. I den simuleringsmetod som behandlas i detta arbete strävar vi efter att separera all kod i ett system i två delar, hårdvarunära och icke hårdvarunära kod.

Interrupt är av naturen hårdvarunära och deras implementation bör därför tillhöra den hårdvarunära koden. CAN-nätverk, seriell kommunikation och annan IO är källan till nästan alla interrupt och hanteras i målsystemet av hårdvarunära kod som har ett väl definierat gränssnitt mot den icke hårdvarunära koden.

I den simulerade miljön finns moduler med samma gränssnitt mot den icke hårdvarunära koden, men implementationen på den simulerande datorn är vitt skild från målsystemets implementation. Det är alltså inte meningsfullt att infoga brytpunkter med målsystemtider i den simulerade hårdvarunära koden eftersom det inte är samma kod.

5.1.1. Synkronisering

För att ändå kunna hålla tidsynkroniseringen för den icke hårdvarunära koden då den simuleras måste målsystemtider för anrop till funktioner genom gränssnittet till den hårdvarunära koden bestämmas. Dessa tider kan fås genom att summera blocktider som fås från kompileringen av den hårdvarunära koden för målsystemet.

Då man summerar kodblockstider på detta sätt fås ett nytt kodblock som kan ha en mycket stor total exekveringstid. Så som tidigare nämnts så är det önskvärt att ha små kodblockstider eftersom det gör att skillnaden mellan processernas målsystemtider kan hållas lägre. Därför kan det vara nödvändigt att bryta upp det stora kodblocket i mindre bitar.

Detta kan ske genom att godtyckligt lägga in ett antal brytpunkter i simuleringssystemets hårdvarunära kod så att den totala summan av brytpunkternas angivna tider motsvarar summan som fåtts från målsystemet. Då detta måste göras manuellt kan det dock bli mycket arbetsamt.

Exekveringstiden för ett anrop till den hårdvarunära koden kan variera. Man kan uppskatta ett medelvärde och använda det som exekveringstid varje gång. Exekveringstiderna kan dock variera ganska mycket och ett medelvärde ger då stora fel. Man kan istället studera den hårdvarunära koden i målsystemet och analysera vilka delar av den som tar mycket tid och sedan söka upp motsvarande delar i simuleringens hårdvarunära kod. Genom att fördela brytpunkterna i simuleringens hårdvarunära kod så att den summerade exekveringstiden varierar på samma sätt som i målsystemet kan ett bättre resultat än med medelvärden uppnås. Då även detta måste göras manuellt kan det bli mycket arbetsamt, det behöver dock bara göras en gång för varje typ av målsystem.

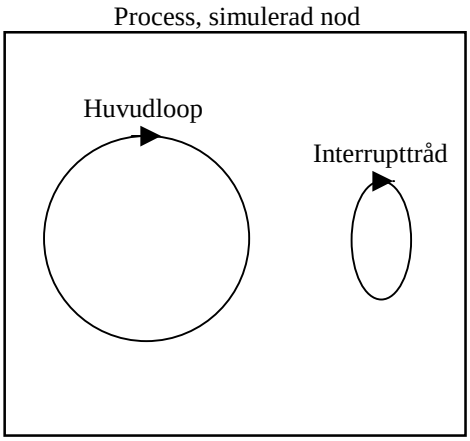
5.1.2. Interrupten förbrukar processortid

Den processortid som interrupten förbrukar i målsystemet syns inte i simuleringen eftersom de tillhör den hårdvarunära koden i målsystemet som saknar en direkt motsvarighet i simuleringen. För att efterlikna den inverkan detta kan ha på koden i noden kan vi skapa simulerade interrupt. I processen som

simulerar noden skapas då en extra tråd, den avbryter huvudloopen med vissa mellanrum och förbrukar lite målsystemtid. Hur ofta dessa avbrott skall ske och hur mycket tid de ska förbruka kan man bestämma genom att studera målsystemet. Denna metod är inte alls exakt utan enbart avsedd för att göra den ökade processorbelastning som interrupten medför märkbar även i simuleringen. För att få en exaktare simulering av interrupten måste man analysera den kod som simulerar hårdvaran och den hårdvarunära koden i den simulerande datorn. Genom att hitta de delar i koden som skulle kunna motsvara ett interrupt i målsystemet kan man förse dessa delar med brytpunkter som uppdaterar den simulerade nodens målsystemtid. Detta blir mycket arbetsamt och exaktheten är fortfarande låg eftersom det huvudsakliga problemet kvarstår, i denna simuleringsteknik emulerar vi inte målsystemens processorer så det hårdvarunära beteendet kan ej studeras.

5.2. Simulering av interrupt

I det simulerade systemet är varje hårdvarunod representerad av en process. Interrupt som tillhör noden simuleras av en tråd som körs i processen. Denna tråd är för det mesta suspenderad och utför inget arbete, med jämna mellanrum aktiveras tråden och triggar ett interrupt. Tråden kan även använda pollning för att upptäcka IO-aktivitet och då trigga ett interrupt.



Figur 10. En simulerad nod med huvudloop och interrupttråd.

Då ett interrupt triggas måste exekveringen av nodens huvudloop avbrytas och interruptets exekvering påbörjas. Hur skall vi då avbryta huvudloopen? Genom att ge interruptet en högre prioritet i synkroniseringen än övriga trådar tillhörande samma nod kommer det att få exekvera då målsystemtiderna jämförs och körtillstånd ges till den tråd som har lägst målsystemtid eller högst prioritet. Då alla trådar inom en simulerad nod (en process) har samma målsystemtid kommer den med högst prioritet att exekveras först, huvudloopen som har lägre prioritet än interruptet blir alltså avbruten.

5.2.1. Simulerade interrupt sker ej omedelbart

Då ett interrupt triggas kan det inte börja exekvera förrän huvudloopen nått en brytpunkt vilket medför att interruptet blir fördröjt. I hårdvaran påbörjas exekveringen nästan omedelbart med hjälp av processorns interrupthanterare vilket medför att svarstiden för ett interrupt kan bli mycket kort. Om

brytpunkterna ligger tätt blir fördröjningen i simuleringen inte så stor och den blir mer verklighetstrogen. Att uppnå de svarstider som interrupt i hårdvaran har är dock i det generella fallet omöjligt.

Den tråd som exekveras då interruptet triggas behöver inte nödvändigtvis tillhöra samma simulerade nod som interruptet. Om den exekverande tråden tillhör en nod som har lägre målsystemtid än noden som interruptet tillhör kommer interruptet ej att få köra förrän dess målystemtid är minst. Denna fördröjning märks dock bara om man studerar noden utifrån. Under tiden andra noder exekverar står målsystemtiden i den suspenderade noden still och interruptet kommer att påbörja sin exekvering utan extra fördröjning i målsystemtid.

5.2.2. Triggning av simulerade interrupt

I hårdvaran triggas ett interrupt av en timer eller av någon yttre händelse. För att efterlikna detta i simuleringen används trådar så som nämnts ovan. Tråden kan med pollning (detta sker utanför den synkroniserade exekveringen) bevaka och upptäcka en förändring i t.ex. ett variabelvärde och då starta koden för interruptet (denna startas i den synkroniserade exekveringen). Om man använder tid för triggning av interrupt bör man tänka på att den simulerande datorns systemtid kan skilja sig från målsystemtiden, och det är oftast målsystemtiden man vill använda.

6. Implementation av prototyp

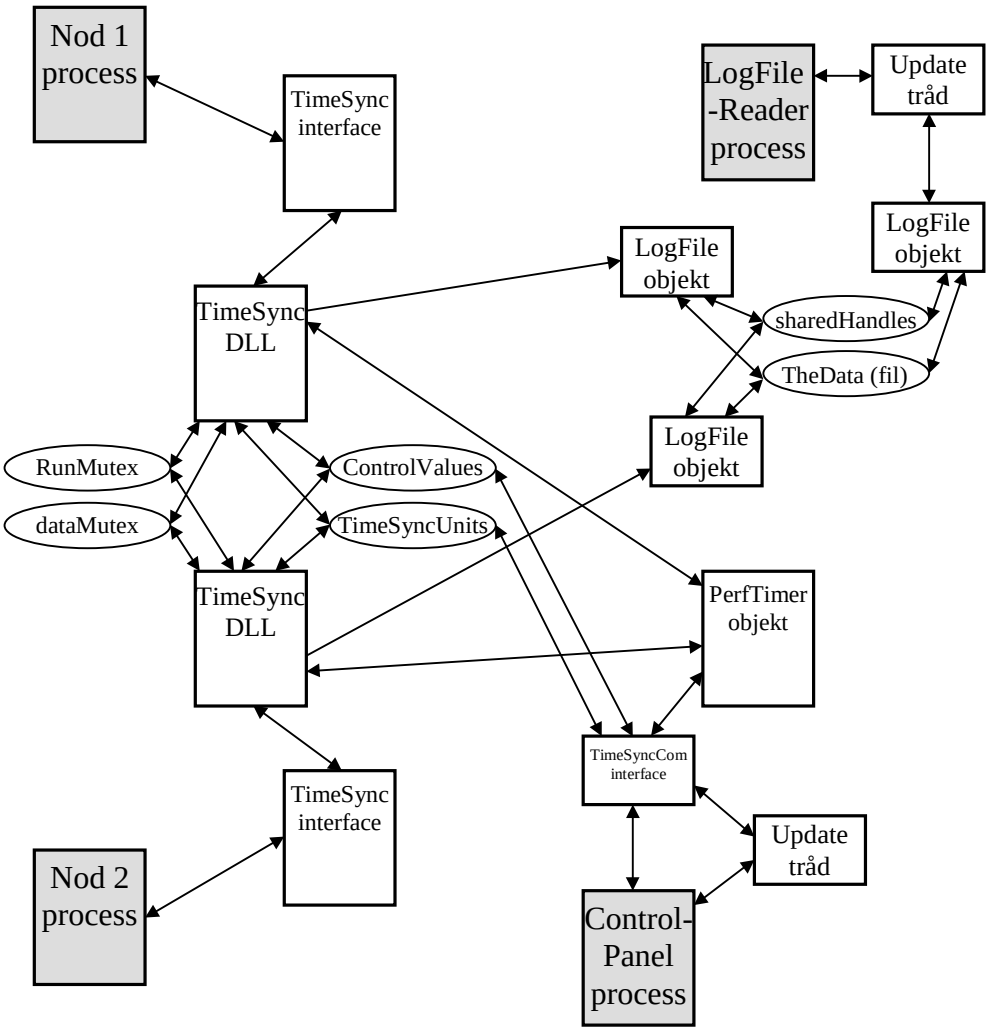
6.1. Inledning

Utifrån den synkroniseringsmetod som beskrivits har jag implementerat en prototyp. Prototypen har skrivits så att den kan utöka det simuleringssystem som CC Systems utvecklat. Prototypen är skriven för operativsystemet Windows NT i en Intel Pentium enprocessormiljö.

6.1.1. Prototypens grundläggande uppbyggnad

I den simulerade miljön körs varje hårdvarunod som en process. Nodernas kod förses med brytpunkter, en brytpunkt består av ett funktionsanrop med förbrukad målsystemtid sedan föregående brytpunkt som argument. Den anropade funktionen summerar nodens inrapporterade tider och jämför summan med de andra processer som körs i systemet. Funktionen blockerar alla processer utom den som har lägst målsystemtid. Om processens målsystemtid är större än simuleringsdatorns systemtid kan funktionen bromsa alla processer så att simuleringen håller en viss hastighet.

6.1.2. Ingående komponenter



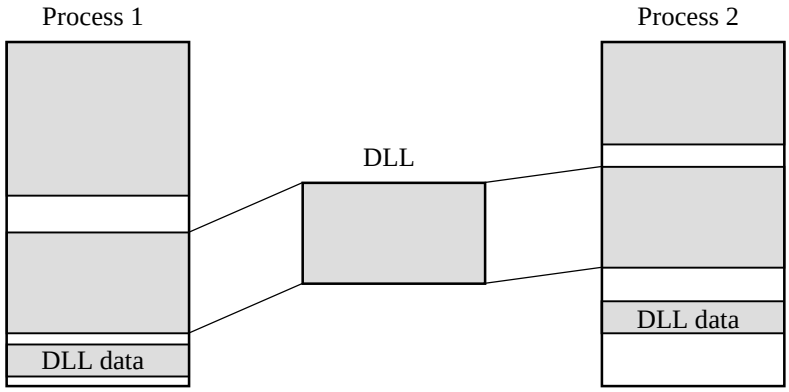
6.2. Specifikt för operativsystemet

6.2.1. Inledning

För att kunna styra de olika processerna som ingår i simuleringen krävs någon form av central enhet som kan få processerna att köra eller suspendera. Processerna måste även ha en gemensam dataarea där deras målsystemtider kan sparas och jämföras. Operativsystemet Windows NT tillhandahåller ett antal funktioner som jag använt för implementationen, de beskrivs kortfattat nedan.

6.2.2. Dynamic Link Library (DLL)

I Windows NT är en DLL ett objekt som kan innehålla både kod och data. DLL:en kompileras separat och kan sedan inkluderas i andra program, antingen genom att inkludera DLL:ens LIB-fil vid kompileringen av programmet eller genom att dynamiskt ladda DLL:en under programmets exekvering.



Figur 11. En DLL som används av två processer.

När en DLL efterfrågas av en process (ett program) för första gången allokerar operativsystemet en plats för dess kod i minnet och lägger den där, för DLL:ens data allokeras utrymme i den anropande processens minnesarea. Operativsystemet mappar sedan in minnet som blev allokerat för DLL:ens kod i processens minnesarea. Om en annan process sedan efterfrågar samma DLL så skapas inget nytt minnesutrymme för DLL:ens kod. Det tidigare allokerade fysiska minnet mappas in i den anropande processens virtuella minnesarea. För DLL:ens data allokeras dock nytt utrymme i den anropande processens virtuella minnesarea.

6.2.3. Delat minne, minnesmappade filer

Genom att använda en DLL kan alltså gemensamma funktioner för kontroll av målsystemtid och körtillstånd lätt göras åtkomliga för alla processer som ingår i simuleringen. Däremot saknas fortfarande möjligheten att utbyta information mellan processerna vilket behövs för synkroniseringen. Genom att i DLL:ens kod skapa handtag till ett delat minnesutrymme kan processerna t.ex. ha en gemensam array med målsystemtider.

I Windows NT skapas delade minnesutrymmen med hjälp av minnesmappade filer. Man kan öppna en fil som en minnesmappad fil och den kan då användas precis som om den vore vanligt minne. Man kan även öppna en

minnesmappad fil som ej finns på disk utan bara tillfälligt existerar i minnet. Minnesmappade filer kan göras åtkomliga från flera processer samtidigt.

6.2.4. Ömsesidig uteslutning, Mutex

Då flera processer delar ett gemensamt minnesutrymme måste man se till att bara en process i taget har möjlighet att ändra i detta. Om ett sådant skydd ej finns kan två processer samtidigt försöka ändra samma data med oförutsägbara konsekvenser. I Windows NT kan man skapa en mutexsymbol. De avsnitt som innehåller kod som använder det delade minnet måste reservera mutexsymbolen för att få exekvera. När koden är färdig med det delade minnet frigör den mutexsymbolen.

6.3. DLL:ens uppbyggnad

6.3.1. Inledning

Jag har valt att implementera prototypen som ett Dynamic Link Library (DLL) som inkluderas av alla processer som ingår i simuleringen. På detta sätt får alla processer i simuleringen tillgång till en gemensam uppsättning funktioner för synkronisering.

6.3.2. Allmän beskrivning

Varje process anropar DLL:ens registreringsfunktion. Då läggs ett fält med processens namn, prioritet och målsystemtid i det minne som delas av alla processer som laddat DLL:en.

Då processen vill låta synkroniseringen mot de andra registrerade processerna börja anropar den DLL:ens startfunktion med sin aktuella målsystemtid som parameter (annars används den aktuella systemtiden). DLL:en innehåller en mutexsymbol som tilldelas den process som har lägst målsystemtid, de övriga processerna blir suspenderade i väntan på mutexsymbolen.

När en process når en brytpunkt anropas DLL:ens waitfunktion med processens använda målsystemtid sedan förra brytpunkten som parameter. Waitfunktionen i DLL:en ser till att den process som nu har lägst målsystemtid får mutexsymbolen och kan exekvera, de andra processerna suspenderas i väntan på mutexsymbolen.

Då en process avslutas anropar den en avregistreringsfunktion i DLL:en som tar bort informationen om denna process ur det delade minnet, de andra processerna tar sedan ej hänsyn till den avregistrerade processen i synkroniseringen.

Om så önskas skrivs en loggfil som innehåller information om när (både målsystemtid och systemklocka) processer registreras, startas, sätts i väntan på andra processer, sätts i väntan på målsystemtid, tillåts köra och avregistreras.

6.3.3. Initiering

Då DLL:en laddas av en process startas dess initieringsfunktion automatiskt (detta sker varje gång en ny process laddar DLL:en). Denna funktion skapar två mutexsymboler, en används för att skydda det data som processerna delar (`dataMutex`) och en används för att enbart ge en process i taget möjlighet att exekvera (`RunMutex`). Sedan initieras de minnesmappade filer som används för data som delas mellan processerna i simuleringen. Om den minnesmappade filen inte finns i systemet så skapas den, om den redan finns öppnas den.

Datastrukturer

Det skapas ett antal minnesmappade filer för att hantera simuleringen. En av de minnesmappade filerna innehåller en array av typen `TimeSyncUnitType`. Där lagras information om varje registrerad process. Ett element i arrayen innehåller:

- Namn
- Prioritet, en integer, högt värde innebär hög prioritet.
- Målsystemtid, nuvarande tid i noden som denna process representerar.
- Förälder, om den registrerade processen egentligen är en tråd eller ett interrupt som tillhör en redan registrerad process finns ett handtag till huvudprocessen här.
- Använd, en flagga som anger om detta element används eller ej.

En annan minnesmappad fil innehåller variabler för kontroll av simuleringen, dessa kallas `ControlValues`:

- Loggfil av/på, en flagga som styr om information skrivs till loggfilen.
- Kontrollerad tid av/på, en flagga som styr om exekveringens hastighet ska följa någon yttre klocka.
- Tidsskala, om kontrollerad tid används kan exekveringens hastighet påverkas genom valet av tidsskala.
- Pause, om denna flagga är sann pausas exekveringen av alla processer.
- Stegvis exekvering, denna flagga gör att exekveringen stannar upp vid varje processbyte.
- Aktiv, anger vilken process som exekverar just nu.

Dessa variabler är åtkomliga för andra program, programmet ControlPanel använder dem för att utifrån påverka exekveringen av de processer som ingår i simuleringen.

Det skapas även en minnesmappad fil med ett objekt av klassen `PerfTimer`. Objektet initieras och används sedan för alla tidmätningar i DLL:en som använder den simulerande datorns systemklocka, alla processer använder sig alltså av samma objekt för tidmätningar för att få enhetliga resultat.

6.3.4. Registrering

Efter att en process som ska ingå i synkroniseringen har laddat DLL:en måste den registreras. Det sker med funktionen `TimeSyncRegisterUnit`. Som inparametrar skickas en sträng med processens namn samt en variabel av typen `long` som anger processens prioritet. Funktionen returnerar ett handtag som används för att identifiera processen vid senare anrop till funktioner i DLL:en.

Då en process registreras läggs ett fält upp i det delade minnet med dess namn, prioritet, målsystemtid och förälder. Denna information används sedan i de jämförelser av målsystemtid som görs för att avgöra vilken process som ska få exekvera.

Om en nod har flera exekverande trådar eller om man vill simulera ett interrupt i en nod kan man registrera flera processer för samma nod. Dessa processer ska då alltid ha samma målsystemtid, eftersom de alla exekverar på samma simulerade nod. Då man registrerar den andra (eller tredje, fjärde osv.) processen som tillhör samma nod anger man den först registrerade processen som förälder. Dessa processer kopplas då samman och kommer alltid att ha samma

målsystemtid. Det finns bara en förälder i en grupp av sammankopplade processer. Om man anger en medlem i en grupp som förälder vid en ny registrering kontrolleras att denna process inte själv har någon förälder, om den har det används denna som förälder även till den nyregistrerade processen.

6.3.5. Start

Då en process vill påbörja synkroniseringen med de andra registrerade processerna anropar den funktionen `TimeSyncStart` eller `TimeSyncStartTime`. Som inparametrar skickas handtaget som returnerades vid registreringen samt processens målsystemtid. Om ingen målsystemtid skickas sätts målsystemtiden till den globala systemtiden vid starttillfället som fås från klassen `PerfTimer`. Om simuleringen körs med kontrollerad tid (realtid) kommer processen alltså ursprungligen att få ungefär samma målsystemtid som de andra processerna i systemet.

Startfunktionen uppdaterar processens målsystemtid i det delade minnet och försöker sedan låsa mutexsymbolen för att få exekvera, `RunMutex`. Processen suspenderas tills mutexsymbolen frigörs om mutexsymbolen är upptagen av någon annan process. Mutexsymbolen kommer frigöras av de andra processerna då de ej längre har den lägsta målsystemtiden.

När startfunktionen får mutexsymbolen anropas DLLens Waitfunktion, där avgörs om/när processen skall få tillstånd att fortsätta exekvera. Processen är nu en del av det synkroniserade systemet.

6.3.6. Brytpunkt, Waitfunktionen

I slutet av startfunktionen och vid de brytpunkter som infogas i nodernas källkod anropas funktionen `TimeSyncWait`. Den hanterar de centrala funktionerna i tidssynkroniseringen. Här jämförs prioriteter och målsystemtider mellan de synkroniserade processerna och här avgörs vilken process som får tillstånd att köra.

Funktionen anropas med två parametrar, handtaget som returnerades vid registreringen samt den målsystemtid som gått sedan förra anropet till funktionen. Först uppdateras värdet för processens målsystemtid i det delade minnet, det ökas med den förbrukade målsystemtiden som fås från inparametrarna. Om processen har en förälder så uppdateras dess målsystemtid istället.

Jämförelse- och vänteloopen

Waitfunktionen går in i en loop som fortsätter så länge som den här processens målsystemtid inte är den lägsta. Inne i loopen jämförs processens målsystemtid med de andra registrerade processernas målsystemtider. Då jämförelse görs uppstår ett antal möjliga fall:

- Processens tid är inte lägst
- Processens tid är lägst och inga andra processer har samma tid
- Processens tid är lägst men det finns andra processer med samma tid

I det första fallet avslutas jämförelserna, detta innebär att processen ska suspenderas. Det andra fallet innebär att processen ska återgå till sin normala exekvering. I det tredje fallet jämförs prioriteterna hos de processer som har samma tid och en ny uppsättning fall uppstår:

- Processens prioritet är inte högst

- Processens prioritet är högst och inga andra processer med samma tid har samma prioritet
- Processens prioritet är högst men det finns andra processer med samma tid som har samma prioritet

I det första fallet avslutas jämförelserna, detta innebär att processen ska suspenderas. Det andra fallet innebär att processen ska återgå till sin normala exekvering. I det tredje fallet kontrolleras om processen just kommit in i loopen eller om den redan gått ett eller flera varv. Om den just kommit in i loopen innebär det att processen ska suspenderas, annars ska processen återgå till sin normala exekvering. Åtgärden i det tredje fallet är till för att låta processer som redan väntat på sin tur köra före processer som just kört.

Byte av aktiv process

Om resultatet av jämförelserna av processernas målsystemtider och prioriteter är att exekveringen av processen ska suspenderas och att en annan process nu ska ta över mutexsymbolen så frigör man mutexsymbolen. Den process som stått på kö i väntan på mutexsymbolen längst tid får den. Efter att mutexsymbolen frigjorts ställer sig processen omedelbart i kö för att återfå den, då symbolen nu är upptagen av en annan process suspenderas processen i väntan på att mutexsymbolen ska frigöras igen.

Den process som nu fått mutexsymbolen har tidigare ställts i kö på samma ställe. När den nu fortsätter genomgår den en jämförelse av tider och prioriteter. Om den nyväckta processen efter jämförelse har den lägsta målsystemtiden får den exekvera, annars suspenderas den igen och nästa process i kön väcks upp. Processer väcks och suspenderas på detta sätt tills processen med lägst målsystemtid hittas. Så länge en process inte har lägst målsystemtid stannar den (mestadels suspenderad) i loopen.

Problem med Windows NT:s schemaläggare

Då en symbol i operativsystemet frigörs och den process som varit suspenderad i väntan på den väcks upp ges processen en dynamisk prioritetshöjning av Windows NT. Detta medför att så fort mutexsymbolen släpps av den först nämnda processen ovan kommer den nyväckta processen att få en högre prioritet i Windows NT:s schemaläggare. Den första processen kommer att avbrytas (pre-emption) innan den hinner ställa sig i kö för mutexsymbolen igen. Det är viktigt att processen som släppt mutexsymbolen så snabbt som möjligt ställer sig i kö för den igen, för att ge den en chans att göra det så har jag lagt till ett `Sleep(0)` kommando direkt efter det att en process återfått mutexsymbolen. Då släpper den nyväckta processen fram processen som släppte mutexsymbolen (eftersom den utsattes för pre-emption så står den först i scheduleringskön) så att den kan ställa sig i kö igen och bli suspenderad i väntan på symbolen[3].

Synkronisering med systemtid

När processen reserverat mutexsymbolen är den redo för att återgå till sin exekvering. Först sker dock ett antal kontroller, vilka kontroller som görs beror på värdena i datastrukturen `ControlValues`.

Om `ControlValues` anger att kontrollerad tid skall användas jämförs processens målsystemtid med den simulerande datorns systemklocka. Om systemklockans tid är lägre än målsystemtiden väntar processen tills tiderna är

[3] Se mer i [1].

lika. På detta sätt fås simuleringen att exekvera i samma hastighet som målsystemet (eller åtminstone i den hastighet som de i koden annoterade målsystemtiderna anger). En förutsättning för att detta ska fungera är att den simulerande datorn exekverar koden fortare än målsystemet så att det finns tid över för att ”vänta in” målsystemtiden.

Då målsystemtiden jämförs med systemtiden multipliceras systemtiden med en faktor kallad `timeScale`. Denna faktor är i normalfallet ett, men om man vill köra simuleringen fortare eller i slowmotion kan man ändra dess värde (värden större än ett medför en ökad exekveringshatighet).

Paus och stegvis exekvering

Det finns två flaggor i `ControlValues` kallade `PauseExec` och `StepExec`. Innan Waitfunktionen avslutas kontrolleras dessa flaggor var och en och ifall någon av dem är flaggad så går processen in i en vänteloop. Då processen väntar stoppas även den globala målsystemtiden. Dessa flaggor används som namnen antyder för att utifrån tillfälligt stoppa exekveringen.

Efter alla kontroller är det processen med lägst målsystemtid och högst prioritet som har reserverat mutexsymbolen och därmed kan fortsätta sin exekvering fram till nästa brytpunkt.

6.3.7. Avregistrering

Då en process avslutas eller vill lämna synkroniseringen måste den avregistrera sig. Då avregistreringsfunktionen anropas tas processens namn och målsystemtid bort från det gemensamma minnet så att processen inte längre ingår i simuleringens tidsjämförelser. Då funktionen avslutas släpps mutexsymbolen så att de andra processerna i simuleringen kan ta över den.

Om en process är förälder till en eller flera andra processer kan den inte avregistreras förrän dessa processer avregistrerats. Om man försöker ges ett felmeddelande. Då en process som har en förälder avregistreras uppdateras deras gemensamma målsystemtid (med det värde som ges som inparameter) innan avregistreringen slutförs.

Om en process avslutas utan att avregistrera sig kommer dess målsystemtid ligga kvar i det gemensamma minnet och eftersom denna tid inte längre uppdateras kommer denna tid snart att vara lägst. Detta medför att alla andra processer kommer vänta på en process som inte längre existerar, systemet blir låst.

6.3.8. Import av DLL:en i processerna

Det finns två sätt att inkludera en DLL i ett program. Den ena är att vid kompileringen av programmet inkludera den LIB-fil som genererades vid kompileringen av DLL:en. LIB-filen är specifik för den kompilator man använder, t.ex. fungerar inte LIB-filer genererade av Visual C++ i C++ Builder. Då LIB-filen inkluderats kan alla DLL:ens funktioner anropas direkt i programmet. När programmet startas laddas DLL:en omedelbart, om den inte finns tillgänglig avbryts exekveringen med ett felmeddelande.

Den andra metoden är att ladda DLL:en dynamiskt under programmets exekvering. DLL:en laddas när den efterfrågas i koden, om den inte finns tillgänglig får programmet ta hand om felhanteringen. När DLL:en laddats finns dess funktioner ej direkt tillgängliga utan man måste skapa pekare till dem.

Denna metod använder inte de kompilatorspecifika LIB-filerna och detta är orsaken till att jag valt denna metod i implementationen.

TimeSync gränssnitt

För att hantera importen av DLL:en i ett program har jag skrivit ett gränssnitt bestående av filerna `TimeSync.h` och `TimeSync.cpp`. För varje funktion i DLL:en finns en motsvarande funktion i gränssnittet.

För att använda en funktion i DLL:en anropar man dess motsvarighet i gränssnittet, denna funktion utför då ett antal uppgifter innan den vidarebefordrar anropet till funktionen i DLL:en.

Först kontrollerar den om DLL:en har laddats in till programmet. Om den inte blivit laddad så laddas den, detta sker bara vid det första funktionsanropet. Sedan kontrolleras om pekaren till den sökta funktionen i DLL:en är definierad. Om den inte är det så används en operativsystemsfunktion för att hitta funktionens adress och en pekare till den definieras. Även detta sker bara en gång, vid det första anropet till denna funktion. När DLL:en är laddad och pekaren till funktionen definierad används funktionspekaren för att anropa funktionen i DLL:en.

Detta kan verka omständigt, men då DLL:en laddats och pekarna definierats (vilket sker vid de första funktionsanropen) så skickar gränssnittet bara vidare parametrarna till motsvarande funktion i DLL:en.

6.4. Viktiga stödklasser

I detta avsnitt beskrivs två klasser som används i alla delar av simuleringssystemet. Först beskrivs `PerfTimer`, en klass som används för att avläsa den simulerande datorns systemklocka med hög precision. Sedan beskrivs LogFile, en klass som gör det möjligt för de parallellt exekverande processerna i det simulerade systemet att dela på en gemensam loggfil.

6.4.1. PerfTimer

PerfTimer är en klass som med hög precision mäter tid med hjälp av den simulerande datorns systemklocka. Tiden kan mätas i förhållande till en referenspunkt, eller så kan man läsa av systemklockans nuvarande värde.

Klassens funktioner

Då klassen initieras sätts en referenspunkt, för att avläsa tiden som gått sedan denna referenspunkt anropas funktionen `halftime`. Referenspunkten kan sättas om med ett anrop till funktionen `start`. Tidmätningen kan stoppas med ett anrop till funktionen `stop`, därefter avläsas tiden mellan start och stop med funktionen `result`. Om man temporärt vill stoppa tidräkningen utan att sedan bli tvungen att sätta en ny referenspunkt (vilket är fallet om man använder stop) kan man använda funktionerna `pause` och `resume`.

För att avläsa systemtiden, som refererar till när den simulerande datorn startades, används PerfTimers funktion `systime`.

Tidsenhet och precision

De funktioner i klassen som returnerar tid gör det i form av ett 64 bitars heltal, definierad som typen `TIMESYNCTIME`. Detta heltal innehåller tiden mätt i pikosekunder (1 ps = 10^{-12} s), samma typ och tidsenhet används i alla delar av synkroniseringssystemet. Denna representation av tid har valts för att den kan lagra stora tal (långa tider, maximalt ca 210 dygn) samtidigt som den har hög upplösning.

Systemklockan som används har dock inte så hög upplösning. Jag använder Windows NT:s PerformanceCounter för att avläsa systemklockan, dess upplösning varierar beroende på simuleringsdatorns hårdvara. För ett system med en Intel Pentium III processor ligger upplösningen strax under en mikrosekund (10^{-6} s).

Användning i DLL:en

I DLL:en används PerfTimer i alla sammanhang där systemtiden behövs. Eftersom alla processer måste ha exakt samma systemtid så har jag gjort så att de alla delar samma objekt av klassen PerfTimer. Då DLL:en laddas för första gången initieras ett antal minnesmappade filer som innehåller det data som processerna behöver dela, däribland ett objekt av PerfTimer klassen. Då detta objekt skapas sätts dess referenspunkt, alla senare tidmätningar (som använder funktionen `halftime`) har denna tidpunkt som tiden noll.

När andra processer sedan laddar in DLL:en skapas inget nytt PerfTimer objekt vid DLL:ens initiering utan de använder det redan existerande objektet i det delade minnet. På detta sätt får alla processer samma referenspunkt för sina tidmätningar.

I DLL:en används systemtiden som jämförelse mot målsystemtiden för att ”bromsa upp” exekveringen, samt för att sätta tidstämplar på det som skrivs till loggfilen. I det första fallet används funktionen `halftime` eftersom man vill ha tidpunkten noll vid simuleringens början. I det andra fallet används funktionen `systime` eftersom den är oberoende av eventuella `start`-, `pause`-, `resume`- eller `stop`kommandon. Då kontrollflaggorna `pauseExec` eller `stepExec` används för att temporärt stoppa simuleringen anropas `pause` och `resume` funktionerna i PerfTimer för att undvika att tiden som `halftime` returnerar växer under stoppet.

6.5. Kontrollfunktioner

6.5.1. ControlPanel

När man använder tidsynkroniserings-DLL:en i sina processer får man nya möjligheter att kontrollera deras exekvering. Andra program kan påverka simuleringens beteende utifrån (och därmed indirekt påverka de synkroniserade processerna) med hjälp av kontrollstrukturen `ControlValues`. Programmet ControlPanel utnyttjar detta och tillhandahåller ett grafiskt användargränssnitt så att användaren kan styra exekveringen.

Användargränssnitt

Kontrollpanelen visar information om de processer som är registrerade i DLL:en och har ett antal knappar för att styra exekveringen.

Högst upp i mitten sitter tre knappar, ”Play”, ”Step” och ”Pause”. Precis som deras namn anger startar och pausar de processerna i simuleringen. Step-knappen låter en process exekvera till nästa brytpunkt och ställer den sedan i pausläge.

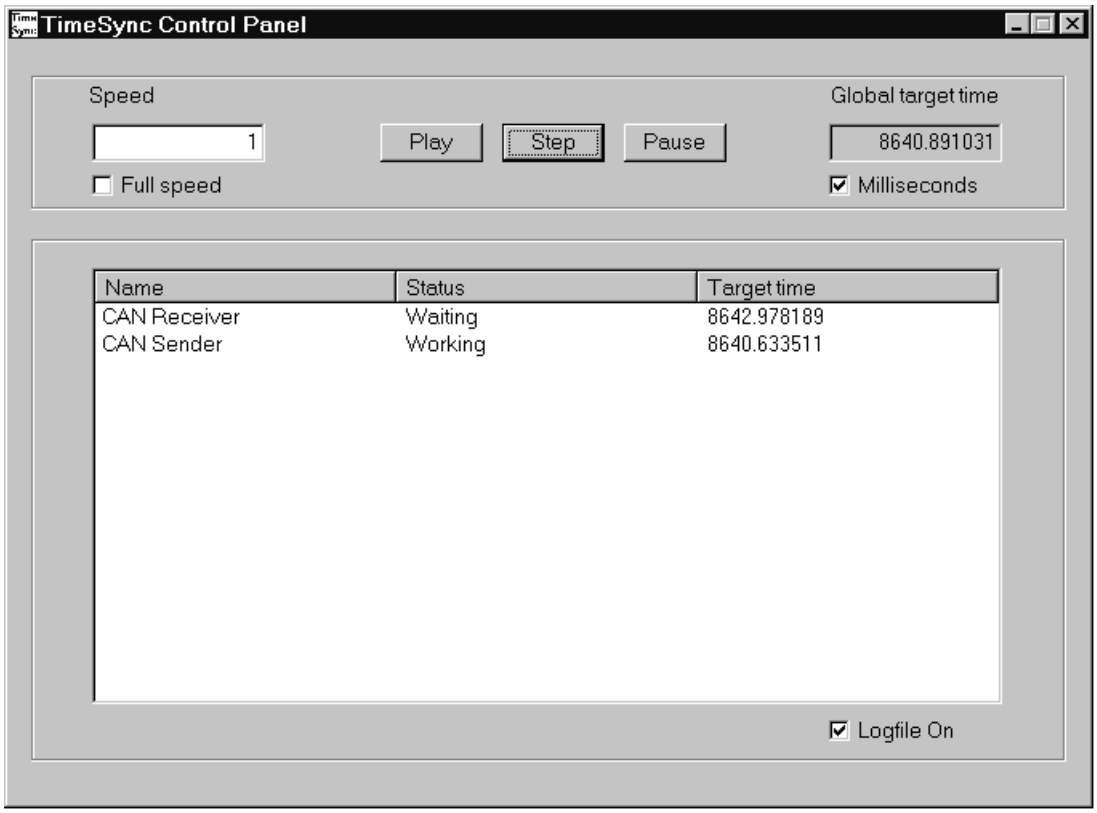
Längst upp till vänster finns textrutan ”Speed”, där anger man vilket värde variabeln `timeScale` ska ha. I normalfallet är dess värde ett vilket innebär att processernas målsystemtider jämförs med systemklockans tid gånger ett. Detta innebär att simuleringen bromsas upp så att den går i ”riktig” tid, detta läge är lämpligt för blandsimulering. Om värdet ”Speed” ändras kan simuleringen fås att gå fortare eller långsammare, värden mindre än ett medför långsammare exekvering och vice versa.

Genom att kryssa för rutan ”Full speed” sätts flaggan `useControlledTime` till falsk. Detta innebär att inga jämförelser mellan målsystemtiderna och systemtiden görs och att simuleringen går i högsta möjliga hastighet. Om ”Full speed” är förkryssad har värdena i textrutorna ”Speed” och ”Global target time” ingen relevans för simuleringen.

Längst upp i högra hörnet visas ”Global target time”, det är systemtiden multiplicerad med faktorn `timeScale`, dvs. den tid som processernas målsystemtider synkroniseras med. Med kryssrutan under ”Global target time” väljer man vilken enhet som skall användas för alla tidsangivelser i kontrollpanelen. Om rutan är förkryssad används millisekunder, annars används simuleringssystemets interna tidsenhet, pikosekunder (10^{-12} s).

Det stora området i mitten av fönstret visar en lista över alla processer som är registrerade till DLL:en. Listan innehåller processernas namn, om de är i väntläge eller är aktiva samt deras målsystemtid.

Längst ner i högra hörnet finns en kryssruta som styr om det skall skrivas information till loggfilen eller inte, denna ruta är i normalfallet ikryssad.



Figur 12. Kontrollpanelens användargränssnitt.

Uppbyggnad av ControlPanel

Kontrollpanelen utför inget arbete själv utan är bara ett skal som presenterar och skickar vidare information. Kommunikationen med DLL:en och bearbetningen av knapptryckningar sköts av en klass kallad TimeSyncCom. Den kontinuerliga presentationen och uppdateringen av processernas status och av systemtiden sköts av en tråd som startas av funktioner i filen updateLoop. Tråden använder också klassen TimeSyncCom för att kommunicera med DLL:en.

Klassen TimeSyncCom

Klassen TimeSyncCom används för att läsa värden ur DLL:ens minnesmappade filer samt för att ändra värden i datastrukturen `ControlValues` och därmed påverka DLL:ens beteende.

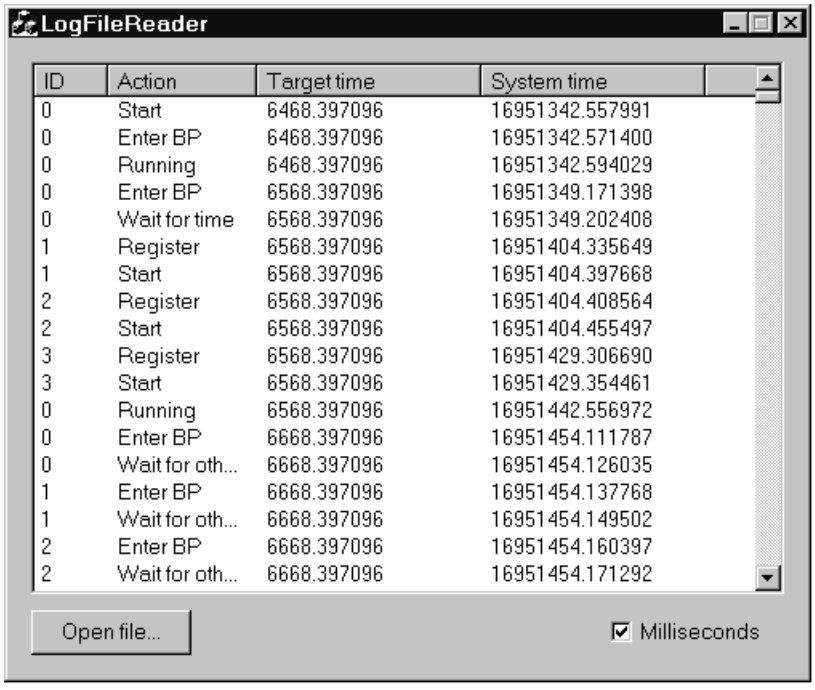
Tråden updateLoop

För att uppdatera listan över processer samt värdet i rutan ”Global target time” kontinuerligt startas en tråd vid initieringen av ControlPanel. Tråden består av en loop som läser värden från DLL:en med hjälp av klassen TimeSyncCom, formaterar utskriften och skriver till fönstret. För att minimera flimmer uppdateras elementen i listan enbart om det skett förändringar av deras värden. Alla element i listan som inte längre har en motsvarande process registrerad i DLL:en raderas.

Loopen innehåller ett Sleep-kommando för att minska dess förbrukande av processortid, uppdateringarna sker med ungefär 300 ms mellanrum.

Loopen fortsätter så länge flaggan `keepRunning` är sann. För att stoppa loopen (och därmed avsluta tråden) finns en stop-funktion som sätter flaggan till falsk. När tråden avslutas sätts en annan flagga, `running`, till falsk. Detta utnyttjas av ControlPanel för att vänta till tråden är helt avslutad innan fönstret stängs.

6.5.2. LogFileReader



Figur 13. LogFileReaders användargränssnitt.

De loggfiler som genereras av DLL:en lagras i ett binärt format och för att läsa dem används programmet LogFileReader. Programmet presenterar loggfilen som en lång lista innehållande processens nummer, händelsen, processens målsystemtid vid händelsen och systemtiden vid händelsen. Användargränssnittet är enkelt och består bara av två knappar. Längst ner i vänstra hörnet finns en ”Open file...”-knapp och längst ner i högra hörnet finns en kryssruta med vilken man kan välja enhet för de tider som presenteras. Om rutan är ikryssad visas tider

i millisekunder, annars visas de i det format som används internt, pikosekunder (10^{-12} s).

6.6. Användning och införande i källkoden

För att köra en simulering där noderna är synkroniserade och exekverar i samma hastighet som i målsystemet måste vi veta hur fort nodernas kod exekverar i sina noder. Synkroniseringsmetoden som beskrivs i detta arbete bygger på att nodernas kod är uppdelad i någon form av block där varje block har en känd exekveringstid på målsystemet. Var ska vi då få denna information?

6.6.1. Information från kompilatorn, framtidsvision

Helst vill vi förstås att informationen om exekveringstiderna ska genereras automatiskt i den utvecklingsmiljö vi använder för att skapa källkoden. Idag finns inte de verktyg som är nödvändiga för att förverkliga detta. Nedan beskrivs hur ett framtida system skulle kunna fungera.

Framtidsvision

Då vi vill prova den nyskrivna koden kompileras den med en kompilator som genererar WCET-information (Worst-Case Execution Time) för den processortyp som finns i målsystemet. Denna information skrivs in som annoteringar i källkoden, efter varje basblock läggs en annotering innehållande exekveringstiden för basblocket på målsystemet. Exekveringstiden som fås från kompilatorn anges i klockcykler, då vi vet vilken klockfrekvens processorn i målsystemet jobbar med är det dock lätt att konvertera klockcyklerna till en annan tidsenhet.

Ett mellanprogram konverterar annoteringarna till brytpunkter för tidsynkroniserings-DLL:en. Den nya koden med brytpunkter kompileras sedan för det simulerande systemet och noden kan då simuleras och köras med samma hastighet som i målsystemet.

Vad saknas?

Idag finns det inga kommersiellt tillgängliga WCET-kompilatorer som genererar annoteringar i koden så som beskrivs ovan. Utvecklingsarbete pågår dock och kompilatortillverkaren IAR-Systems har en prototyp på gång.

Det program som konverterar kompilatorns annoteringar till rätt tidsenhet och gör dem till brytpunkter anpassade för tidsynkroniserings-DLL:en kan relativt lätt skrivas den dag då formatet på kompilatorns annoteringar blir känt.

6.6.2. Tider, tidtagning

Eftersom det ännu inte är möjligt att få exekveringstiderna automatiskt får man i väntan på bättre verktyg ta till manuella mätningar eller uppskattningar. Att mäta exekveringstider på målsystemet är dock ganska svårt. En metod är att använda en av målsystemets digitala utgångar (om det har några) och i koden lägga till kommandon som ändrar utgångens värde vid början och vid slutet av det kodblock man vill tidsbestämma. Man kan då mäta tiden mellan utgångens förändringar med ett oscilloskop. Mätningen kommer inte att bli exakt eftersom koden som ändrar utgångens värde tar lite tid, felet borde dock vara litet.

Att mäta exekveringstiderna på målsystemet motverkar flera av de goda effekterna som man vill uppnå med simulering. För att göra mätningarna måste man ha tillgång till ett målsystem och köra koden på detta. Att sätta upp sitt målsystem och köra kod på det medför ofta mycket arbete. Dessutom måste man

göra nya mätningar varje gång man gör ändringar i koden. Då en nod har en version av sin kod som är färdig eller inte ändras så ofta kan det dock vara värt besväret att mäta upp dess exekveringstider. Då detta är gjort kan denna nod simuleras och köras i rätt hastighet tillsammans med andra noder som är under utveckling.

6.6.3. Mer om brytpunkter

Det finns två olika typer av brytpunkter som kan användas i nodernas källkod då tidsynkroniserings-DLL:en ska användas. Den första typen består av ett anrop till funktionen `TimeSyncWait` i filen `timesync.cpp`. Funktionen tar två parametrar, handtaget för den registrerade processen och ökningen i målsystemtid sedan förra anropet till funktionen.

Om man får exekveringstiderna från en kompilator så kommer brytpunkterna att ligga efter varje basblock i koden. Ett basblock är vanligtvis mycket kort och antalet anrop till `TimeSyncWait`-funktionen blir mycket stort. Vid ett anrop till `TimeSyncWait`-funktionen görs ett ganska stort antal jämförelser och anrop till ytterligare funktioner vilket tar en del tid. Mängden instruktioner som utförs av `TimeSyncWait`-funktionen kommer vida överstiga det fåtal instruktioner som ett basblock mellan två brytpunkter utför. Detta kan leda till att tidsynkroniseringen skapar en overhead som blir oacceptabel.

För att lösa detta problem finns en annan typ av brytpunkt. Denna brytpunkt består av ett makro som är definierat i filen `timesync.h`. Istället för att göra ett anrop direkt till `TimeSyncWait` skriver man makrots namn, `TSbreak`, med samma parametrar som för `TimeSyncWait`. Makrots namn byts vid kompileringen ut mot ett antal instruktioner.

Idén med makrot är att undvika att `TimeSyncWait` anropas vid varje brytpunkt. Vid brytpunkten ökas istället en för processen global variabel, `ttimeused`, med den tid som ges av inparametern. Sedan jämförs `ttimeused` med en annan global variabel, `ttimestop`.

Om `ttimeused` är större än `ttimestop` anropas `TimeSyncWait`, annars görs ingenting och processens exekvering fortsätter. På detta sätt låter man processen exekvera i minst `ttimestop` tidsenheter innan `TimeSyncWait` anropas och de brytpunkter som passeras under den tiden består bara av ett fåtal instruktioner utan funktionsanrop.

När sedan `ttimeused` blir större än `ttimestop` anropas `TimeSyncWait` med de vanliga parametrarna samt med pekare till de globala variablerna. Dessa pekare används för att nollställa `ttimeused` och för att sätta `ttimestop` till ett lämpligt värde.

Denna metod minskar tidsynkroniseringens overhead men sänker samtidigt noggrannheten eftersom tiden mellan anropen till `TimeSyncWait` ökar och därmed även tiden mellan de punkter där processbyten är möjliga. Genom att välja ett lämpligt värde på `ttimestop` kan man hitta en fungerande kompromiss. Värdet är satt till 10 ms i standardfallet.

När makrot används måste de globala variablerna initieras, därför måste pekare till dem även skickas till DLL:ens Start- och Unregister-funktioner.

7. Resultat och testkörningar

7.1. Testprogram

För att testa implementationen av tidsynkroniseringen har jag skrivit ett antal testprogram. Testprogrammen består av en eller flera noder som registrerar sig i tidsynkroniserings-DLL:en. Noderna exekverar parallellt och deras exekvering kontrolleras av DLL:en. Efter körning har jag studerat loggfilen som skrevs under exekveringen.

7.1.1. Test 1, två processer

Det första och enklaste testprogrammet består av två noder. Varje nod importerar DLL:en, registrerar sig som en process i DLL:en, anropar DLL:ens start-funktion och går sedan in i en loop. Loopen innehåller en tidskrävande beräkning och en brytpunkt. Loopen körs ett förutbestämt antal gånger, sedan avregistreras processen från DLL:en och nodens exekvering avslutas.

De två noderna körs samtidigt och då kan man studera hur DLL:en växelvis ger dem tillstånd att exekvera beroende på deras målsystemtider. Om en den ena nodens brytpunkt anger en mindre mängd förbrukad målsystemtid varje loopvarv än den andra ser man hur den noden får köra mer för att målsystemtiderna för de båda noderna skall hållas på samma nivå.

Den förbrukade målsystemtiden som anges i brytpunkten måste vara större än tiden som beräkningen förbrukar i den simulerande datorn (eftersom en förutsättning för att simuleringen skall fungera är att den simulerande datorn är snabbare än målsystemet). Genom att ändra beräkningen så att den tar längre eller kortare tid medan man anger samma mängd förbrukad målsystemtid i brytpunkten efteråt kan man se hur olika belastning på simuleringsdatorn påverkar tidsynkroniseringen.

Simuleringsdatorns beräkningstid har enbart betydelse då man kör en simulering där målsystemtiderna synkroniseras med systemklockan, om man kör utan denna funktion inkopplad körs allt rakt på i simuleringsdatorns tempo, synkroniseringen mellan processernas målsystemtider hålls dock fortfarande.

7.1.2. Test 2, simulerat interrupt

Detta testprogram körs tillsammans med noderna från test 1. Med jämna mellanrum registrerar programmet en ny process som har en av processerna från test 1 som förälder. Den registrerade processen har högre prioritet än sin förälder och får därför exekvera före denna vid nästa brytpunkt.

Efter att programmet registrerat sig anropar det start-funktionen, utför sitt arbete (i detta fall ett anrop till funktionen Sleep) och når sedan en brytpunkt. Efter brytpunkten avregistreras processen.

Programmet väntar sedan en given tid och registrerar, kör och avregistrerar sig sedan på samma sätt igen.

7.1.3. Test 3, multipla trådar

Detta program skapar ett en kontinuerligt exekverande huvudtråd och en timer som periodiskt startar nya trådar med korta arbetsuppgifter. Huvudtråden registrerar sig, anropar start-funktionen och går in i en loop. Loopen utför en arbetsuppgift (i detta fall ett anrop till funktionen Sleep) och har efter det en brytpunkt.

De trådar som startas av timern registrerar sig med huvudprocessen som förälder. De utför en kort arbetsuppgift, når en brytpunkt och avregistrerar sig sedan.

Det intressanta här är att studera vad som händer då flera trådar hinner startas av timern innan huvudtråden når nästa brytpunkt.

7.2. Resultat

7.2.1. Vad man kan se i loggfilen

De flesta resultat jag redovisar har jag fått genom att studera loggfilen som skrivs av DLL:en då simuleringen körs. I loggfilen skrivs element som innehåller processnummer, händelse, målsystemtid och tidstämpel (simuleringsdatorns systemtid).

Först i loggfilen syns hur noderna registrerar sig och hur de anropar start-funktionen. Sedan kommer element som genererats av brytpunkterna i noderna. I slutet av start-funktionen anropas alltid Wait-funktionen (den som anropas i brytpunkterna) så efter elementet ”Start” kommer alltid elementet ”Enter BP” (andra aktiva noder kan dock klämma in ett par element där emellan).

Då en nod går in i en brytpunkt skrivs elementet ”Enter BP” till loggfilen. Genom att jämföra tidstämpeln på elementet ”Enter BP” med tidstämpeln på elementet i loggfilen som skrivs alldeles innan en nod kör sitt kodblock kan man se hur lång tid kodblocket tog att exekvera på simuleringsdatorn.

Man kan även jämföra målsystemtiderna på dessa element för att se hur mycket som adderades till målsystemtiden efter kodblocket.

Om det finns fler processer registrerade och den process som just exekverat inte längre har lägst målsystemtid blir nästa element i loggfilen ”Wait for others...”. Nästa element i loggfilen skrivs av den process som efter jämförelse har visats vara nästa på tur för exekvering. Genom att studera tidstämplarna här kan man se hur lång tid jämförelserna och ett eventuellt överlämnande av mutexsymbolen tar. Mellan elementen ”Enter BP” och ”Wait for others” har alla registrerade processers målsystemtider jämförts och beslutet om vilken som blir nästa exekverande process tagits. Mellan elementen ”Wait for others” och nästa element i filen har mutexsymbolen övergetts av den första noden och tagits över av den nod som nu skall exekvera, ett processbyte har skett. Om elementet ”Wait for others” inte finns där skall processen fortsätta exekvera och inget processbyte behöver ske.

Om man simulerar med kontrollerad exekveringshastighet blir nästa element i loggfilen av typen ”Wait for time”. DLL:en väntar då tills simuleringsdatorns tid (systemtiden med tidsfaktor) överensstämmer med nodens målsystemtid. Alldeles innan noden sedan lämnar brytpunkten och exekverar nästa kodblock skrivs elementet ”Running” till loggfilen.

Om man studerar ett antal efterföljande element som skrivits i loggfilen av en nod kan man hitta mer information. Genom att söka upp ett element av typen ”Running” och jämföra det med nästa ”Running”-element från samma nod kan man se hur mycket målsystemtiden och systemtiden ökat mellan loggpunkterna. Om simuleringen körs med synkronisering mot systemklockan med tidsfaktor ett bör dessa tider ha ökat ungefär lika mycket.

7.2.2. Resultat från test 1, två processer

Efter att ha studerat loggfiler från körningar med noderna i test 1 kan man se att tidsynkroniserings-DLL:en fungerar som avsett. Noderna exekverar växelvis och deras målsystemtider hålls hela tiden på samma nivå. Målsystemtiderna ökar även i samma tempo som den simulerande datorns systemklocka och man kan se hur processerna tvingas vänta för att denna synkronisering ska kunna hållas.

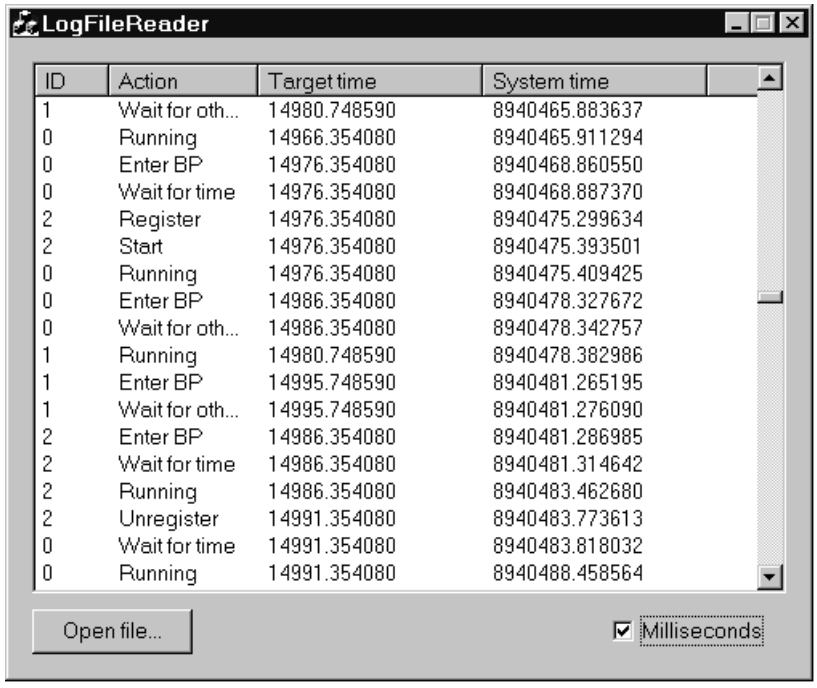
Då man studerar tidsåtgången för de olika momenten som ingår i en brytpunkt ser man som väntat att de blir ungefär lika vid varje körning av simuleringen. Det som förändrar deras värde är antalet registrerade processer.

Med två registrerade processer blir tidsåtgången för jämförelsen av processernas målsystemtider ungefär 27 µs då processen som startar jämförelsen är den som har lägst målsystemtid. Tiden sjunker till ca 11 µs om processen inte har lägst målsystemtid eftersom den då hoppar ur jämförelseloopen tidigare.

Tidsåtgången för att överlämna mutexsymbolen till en annan process varierar mer, för det mesta tog det ca 32 µs men ibland går tiden upp till över 190 µs. Vid de tillfällen tiden blir längre beror det antagligen på att Windows NT schemalägger en annan process som inte ingår i simuleringen i skarven där processbytet sker.

7.2.3. Resultat från test 2, två processer och ett simulerat interrupt

I detta test kördes två noder precis som i test 1 men dessutom kördes en nod med ett simulerat interrupt. I loggfilen syns hur processen för interruptet registreras, hur det tillåts exekvera före sin förälder (process 0 i detta fall) på grund av sin högre prioritet och hur det efter sin exekvering avregistrerar sig.



Figur 14. Loggfil från Test 2, två processer och ett simulerat interrupt.

I figuren ovan visas en bit av loggfilen från en sådan testkörning. Processerna med ID-nummer 0 och 1 är processerna från test 1 och processen med ID-nummer 2 är det simulerade interruptet, process 0 är förälder till process

2. I figuren ser man att process 2 registrerar sig då process 0 står och väntar på att få köra. Eftersom process 0 är redan valts till att köra körs den först, när den når sin brytpunkt har process 1 lägst målsystemtid och får köra. När process 1 når sin brytpunkt har process 0 och process 2 lägst målsystemtid, process 2 har högst prioritet och får därför köra.

7.2.4. Resultat från test 3, multipla trådar

Här var det intressanta att kontrollera att tidsynkroniserings-DLL:en klarade av att synkronisera flera trådar inom en process, inte bara att synkronisera separata processer. Genom att studera loggfilen från simuleringen kunde jag se att det fungerade precis som då man använder separata processer. Tiden det tar i brytpunkten för att jämföra processer varierar beroende på hur många processer som är registrerade just då men är i samma storleksordning som i test 1. Tiden för överlämning av mutexsymbolen är densamma som i test 1.

Då testprogrammets timer skapar och startar upp flera nya trådar innan huvudprocessen når sin nästa brytpunkt ser man i loggfilen hur de registreras direkt och sedan körs i turordning då huvudprocessen senare når sin brytpunkt.

7.3. Overhead

Då noderna förses med brytpunkter för tidsynkronisering ökar givetvis mängden arbete för simuleringsdatorn. Hur stor del av processorns tid kommer att användas för att köra kod för synkronisering? Här får man bara mäta den tid som verkligen används för jämförelser, processbyten och andra aktiva handlingar, mycket processortid kommer att användas i brytpunkterna för att vänta in systemtiden. Detta är dock en önskad effekt och tillhör inte det man kan kalla overhead.

Om man kör synkroniserad simulering men utan synkronisering mellan målsystemtid och simuleringsdatorns systemtid kan all **aktiv** tid förbrukad i brytpunkterna betraktas som overhead. Med aktiv tid menas den tid som går från det att en nod går in i en brytpunkt till det att någon nod lämnar sin brytpunkt och åter börjar exekvera. Om en nod går in i en brytpunkt och får vänta för att en annan nod exekverar beror detta på att de använder samma processor, inte att synkroniseringen stjäl tid.

Genom att studera loggfilen från en simulering med två processer (enligt test 1 ovan) som körts i ”full fart” (utan synkronisering mot systemtiden) kan man få mätvärden på förbrukad tid i brytpunkter. Då jag gjorde detta såg jag att tiderna varierade beroende på om det skedde ett processbyte eller ej, men tiden var alltid lägre än 40 μs. Om man antar att anropen till Waitfunktionen ligger med i genomsnitt 5 ms mellanrum (målsystemtid) i koden gör varje nod 1/0,005 = 200 anrop per sekund. Antagandet ligger nära sanningen om metoden med brytpunkter i form av makron används med `ttimestop` = 5 ms (beskrivs ovan i avsnittet ”Mer om brytpunkter”).

Med detta antagande om 200 brytpunkter/sekund och den observerade maximala exekveringstiden för en brytpunkt ca 40 μs kan man beräkna att för två noder används:

2*200*40 = 16000 μs = 0,016 s

av simuleringsdatorns processortid till synkroniseringsoverhead under simuleringen av en sekund målsystemtid. Hur mycket en sekund målsystemtid motsvarar i simuleringsdatorns processortid beror på hur snabba de två nodernas målsystem är, men eftersom simuleringsdatorn antas vara mycket snabbare än

målsystemen bör processortiden vara mycket mindre än en sekund. Om man vill köra simuleringen i ”realtid” är en overhead i den storleksordning som beräknats ovan helt accepterbar.

7.4. Windows schemaläggare

De processer som ingår i den synkroniserade simuleringen schemaläggs av synkroniserings-DLL:en så att de exekverar i rätt ordning på en processor. Hela simuleringssystemet körs dock under Windows NT som sköter den verkliga schemaläggningen av **alla** processer i simuleringsdatorn och styr hur och när de får tillgång till systemets processor.

Simuleringen kommer alltså att bli avbruten för att köra andra uppgifter och det finns inte stora möjligheter att kontrollera när detta sker. Det bästa man kan göra är att se till att så få processer som möjligt är aktiva utöver de som ingår i simuleringen.

Under utvecklingen av DLL:en hade jag först problem med långa väntetider och låsningar då mutexsymbolen skulle överlämnas från en process till en annan. Efter att noggrannare ha studerat hur Windows NT:s hantering av prioritetsnivåer i scheduleringen fungerar kunde problemen lösas.

Windows NT har fyra prioritetsklasser, Idle, Normal, High och Real-time. De flesta processer ligger i klassen Normal, processerna som tillhör simuleringen ligger i Normal- eller High-klassen. Inom dessa två klasser finns ett antal nivåer som används av Windows NT för att dynamiskt höja och sänka en process eller tråds prioritet beroende på vad den gör för tillfället.

Då en process just blivit väckt efter att ha väntat på ett objekt, t.ex. en mutexsymbol, höjs dess prioritet. Detta medför vissa problem då en process som tillhör simuleringen skall överlämna mutexsymbolen till en annan process. Så fort den första processen släpper mutexsymbolen tas den av en annan process och den andra processens prioritet höjs. Eftersom den andra processen nu har högre prioritet än den första så utförs omedelbart en preemption av den första processen.

Det är bra att den andra processen tillåts exekvera omedelbart men det är även viktigt att den första processen tillåts ställa sig i kö för att återfå mutexsymbolen så snabbt som möjligt. Eftersom den andra processen nu har en högre prioritetsnivå kan det dröja innan den första processen tillåts köra och kan ställa sig i kö igen.

För att lösa detta lade jag in ett Sleep(0) kommando efter det att en process nyss återfått mutexsymbolen. Detta får scheduleraren att släppa fram andra processer vilket ger den första processen den chans den behöver för att ställa sig i kö för mutexsymbolen. En annan möjlig lösning är att tillfälligt stänga av Windows NT:s dynamiska prioritetsförändringar då processbytet sker.

8. Framtida utökningar

8.1. Flera simulerande datorer i nätverk

För stora målsystem med många noder med hög processorhastighet kan det bli svårt för en PC att simulera alla noder i full hastighet samtidigt. Det vore därför önskvärt att kunna dela upp simuleringen på flera datorer. Dessa datorer måste då kommunicera med varandra så att de kan bibehålla synkroniseringen mellan processerna.

Detta problem har inte behandlats i detta arbete men en senare utökning borde vara möjlig. Genom att utnyttja funktionen ExportHandles i DLL:en kan man komma åt all information om de registrerade processerna och med en extra modul som utbyter denna information med de andra simulerande datorerna via t.ex. TCP/IP kan borde problemet kunna lösas.

9. Sammanfattning och slutsatser

Målet med detta arbete var att finna en modell för hur man kan förbättra en existerande simuleringsteknik så att den blir tidsexakt, samt att implementera en prototyp som utökar den tidigare simuleringsmiljön så att den klarar tidsexakt simulering.

Genom att införa brytpunkter i källkoden blev det möjligt att ta kontroll över nodernas exekvering utifrån och att synkronisera dem. Hur exakt denna synkronisering blir beror på hur tätt brytpunkterna läggs i koden. Det mest önskvärda är att kompilatorn för målsystemet annoterar koden och då kan man lägga brytpunkter efter varje basblock. Detta kan dock vara för tätt med tanke på den overhead som brytpunkterna genererar. Enligt de tester jag gjort är dock inte brytpunkternas overhead så stor och man kan även utnyttja de möjligheter som finns för att se till att varje brytpunkt inte innebär ett funktionsanrop (vilket minskar overhead). Exaktheten i synkroniseringen kan alltså hållas hög.

Brytpunkterna medför även att man utifrån kan bromsa exekveringen. Detta har utnyttjats så att nodernas målsystemtid kan synkroniseras med simuleringsdatorns systemklocka. Detta fungerar mycket väl och gör att blandsimulering nu blir möjlig med simulerade noder som kör i ”verklig tid”.

Synkroniseringsmetoden fungerar väl på den icke hårdvarunära koden men på grund av simuleringsmodellens uppbyggnad och skillnader i hårdvaran uppstår problem då interrupt skall hanteras. Interrupt är av naturen hårdvarunära och är svåra att få in i simuleringsmodellen. Ett antal lösningar för att efterlikna interrupt diskuteras men tyvärr finns ingen lösning med tidsexakthet.

En annan begränsning för exaktheten i simuleringen är operativsystemet Windows NT. I detta arbete antas att noderna simuleras i processer med hög prioritet för att undvika avbrott i exekveringen av andra processer utifrån. Detta hjälper dock inte mot avbrott från viktiga funktioner i Windows NT. Efter att ha studerat många loggfiler med resultat från testkörningar verkar dock effekterna av dessa avbrott vara mycket begränsade. Eftersom simuleringen körs i skurar med mycket väntan emellan (simuleringsdatorn är ju mycket snabbare än målsystemet) hamnar oftast avbrotten då simuleringen väntar och de är så korta att de inte skapar problem.

När de kompilatorer som kan generera annoteringar med WCET-estimeringar (Worst-Case Execution Time) i källkoden kommer så tror jag att denna metod för synkronisering av noder under simulering kommer bli ett uppskattat och lättanvänt verktyg. Innan brytpunkterna kan genereras automatiskt och så länge målsystemtider måste mätas på målsystemet kommer kanske användningen vara mer begränsad. För utveckling och tester där man önskar synkronisering mellan noderna men där exaktheten inte behöver vara så hög är dock prototypen mycket användbar redan nu.

10. Källor

1. Custer, Helen, 1993: *Inside Windows NT*, Redmond: Microsoft Press

2. Hansson, Jörgen, 1999-02-26, *Simuleringsteknik*, Uppsala: CC Systems AB

3. Jones, Michael B. m.fl. , 1999, *Results from a Latency Study of Windows NT (2001-03-06)*
   http://research.microsoft.com/~mbj/papers/tr-98-29_abstract.html

4. Muchnick, Steven S. , 1997: *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers

5. Ohlsson, Gunnar, 2000-11-07, *Simuleringsteknik 1.0*, Uppsala: CC Systems AB

6. Richter, Jeffrey, 1994: *Advanced Windows NT*, Redmond: Microsoft Press