Realtidsegenskaper hos Windows NT

Björn Rudin

LiTH-ISY-EX-2027
990329

Realtidsegenskaper hos Windows NT

Examensarbetet utfört i Systemvetenskap
Vid Tekniska Högskolan i Linköping
av

Björn Rudin

Reg nr: LiTH-ISY-EX-2027

Handledare: Erik Dyrelius (SAAB Combitech Software AB)
Examinator: Prof. Lars Nielsen

Linköping 29 mars 1999

Avdelning, Institution
Division, department

Department of Electrical Engineering
Systemvetenskap

Datum
Date

1999-04-14

Språk
Language

☒ Svenska/Swedish
☐ Engelska/English

☐ _____

Rapporttyp
Report: category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport

☐ _____

ISBN

ISRN

Serietitel och serienummer
Title of series, numbering

ISSN

LiTH-ISY-EX- 2027

URL för elektronisk version

Titel
Title

Realtidsegenskaper hos Windows NT

Författare
Author

Björn Rudin

Sammanfattning
Abstract

Windows NT används idag oftast på arbetsstationer och servrar man kan dock se ett ökat intresse för att använda NT i inbyggda system eller i realtidssystem. En undersökning av NT:s lämplighet som realtidssystem har utförts på SAAB Combitech Software AB och presenteras i denna rapport.

Det är inte möjligt att använda NT för hårda realtidstillämpningar då inga tidsgränser på systemanropen finns. Algoritmer som inte är deterministiska används även. Ett systems uppträdande kan även påverkas av installerade drivrutiner. En process basprioritet har stor inverkan på dess känslighet för belastning. Ett genomtänkt användande av prioriteter kan minska denna inverkan.

Genom lämplig dimensionering av ett systems prestanda kan NT användas till mjuka realtidssystem. Ett stort antal mätningar av beteendet hos olika systemmekansismer har utförts vid olika prioritetsnivåer och belastningar. Dessa mätningar ger en bas för dimensionering av ett mjukt realtidssystem som använder Windows NT.

Nyckelord
Keywords

Windows NT, Realtid, Realtidsoperativsystem

Avdelning, Institution
Division, department

Department of Electrical Engineering

Systemvetenskap

Datum
Date

1999-04-14

Språk
Language

☒ Svenska/Swedish
☐ Engelska/English

☐ _____

Rapporttyp
Report: category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport

☐ _____

ISBN

ISRN

Serietitel och serienummer
Title of series, numbering

ISSN

LiTH-ISY-EX- 2027

URL för elektronisk version

Titel
Title

Real Time Properties of Windows NT

Författare
Author

Björn Rudin

Sammanfattning
Abstract

Windows NT is mainly used on workstations or servers. An increasing interest in the usage of NT in embedded and real time systems can be seen. This study of the feasibility of using NT in this kind of applications has been carried out on SAAB Combitech Software AB and is presented in this report.

It’s not possible to use NT for hard realtime applications due to the lack of time bounds on systemcalls.Non-deterministic algorithms is also used in the kernel. The basepriority of an process has an big impact on the sensibility for systemload. With the use of carefully crafted assignments of prioritylevels the systems immunity against system-load can be increased.

Through dimensioning of a systems hardware NT can be used for soft real time applications. A large number of tests on different system mechanisms has been carried out. Theese tests is a good base for dimensioning of soft realtime systems using Windows NT.

Nyckelord
Keywords

Windows NT, Real time, Real time operating system

Sammanfattning

Denna rapport beskriver en utvärdering av Microsoft Windows NT med avseende på realtidsegenskaper.

Ett realtidssystem är ett system där tidskrav är satta på samtliga operationer. Det kan till exempel röra sig om maximal fördröjning i olika former av reglersystem eller krav på reaktionsförmåga i mätsystem. Ett realtidssystem måste vara deterministiskt, det vill säga det måste vara möjligt att förutsäga en maximal tidsåtgång på varje funktion i operativsystemet.

Traditionellt sett har speciella så kallade realtidsoperativsystem används i dessa sammanhang. Ett par exempel på realtidsoperativsystem är pSOS från Integrated Systems och OSE från Enea.

Windows NT designades för att vara ett operativsystem för arbetsstationer det skulle även vara enkelt att anpassa till nya plattformar och marknader utan att för den skull ge avkall på prestanda. Flexibiliteten syns i den flerlagers arkitektur som valts för NT, mycket energi har lagts på att abstrahera bort t ex hårdvaran. Man ser även att vissa eftergifter fått göras för att möta prestandakraven. Många drivrutiner exekverar i kärnans minnesutrymme vilket ger högre prestanda men kan leda till stabilitetsproblem.

De algoritmer som valts i NT har valts för att ge goda prestanda och inte för att vara deterministiska. Man har valt att använda algoritmer som förbrukar mindre systemresurser än motsvarande deterministiska algoritmer, detta för att öka prestanda.

En mängd praktiska tester har utförts både på obelastat och belastat system, testerna har även utförts på olika prioritetsnivåer. Resultaten bekräftar resultaten från studierna av algoritmerna.

Slutsatsen blir att Windows NT inte är lämpligt att använda i hårda realtidssystem utan att det möjligtvis kan användas i mjuka realtidssystem men det finns inget sätt att på förhand bedöma om ett system fungerar eller ej, man måste göra praktiska försök på målplattformen. Man kan inte heller efter tester säga något säkert utan enbart att det antagligen oftast fungerar. Rent praktiskt kan man öka sannolikheten för att systemet fungerar genom överdimensionering av prestanda och minne.

Då så många drivrutiner exekverar i kärnans minnesutrymme så kan även en skenbart harmlös uppgradering av en drivrutin ändra systemets beteende och medföra att ett tidigare fungerande system slutar fungera.

Sammanfattning

1 Introduktion

I detta kapitel beskrivs det övergripande målet för arbetet samt varför det är intressant att studera just detta område. Ur det ganska vagt definierade målet tas en precist formulerad problemformulering fram. Det är detta problem som behandlas i examensarbetet.

1.1 Bakgrund

Traditionellt har Windows NT i första hand använts som operativsystem för arbetsstationer och servrar men på senare tid så har efterfrågan på NT som operativsystem i olika styr- och regler-tillämpningar ökat.

Varför vill man då använda NT som operativsystem för inbyggda tillämpningar? De fördelar man främst ser är:

Billig hårdvara: det är relativt sett billigt att köpa ett PC-kort för inbyggnad jämfört med en annan mer ovanlig hårdvaruplattform.

Billigt operativsystem: NT är relativt billigt jämfört med många RTOS då man inte behöver någon speciell utvecklarlicens för att komma igång utan den enda kostnaden är licenskostnaden per levererad enhet.

Bra utvecklingsmiljö: de utvecklingsmiljöer som erbjuds till NT-plattformen av t ex Borland och Microsoft ger utvecklaren ett större stöd än de flesta utvecklingsmiljöer till de kommersiella RTOSen.

Standard: många företag har idag standardiserat på Microsoft produkter i sina datorsystem för att man vill ha en så homogen datormiljö som möjligt. Om man då kan ersätta de RTOS man har i företagets inbyggda system med NT så kan man dra nytta av den kompetens man byggt upp vad gäller underhåll och utveckling.

Innan man kan använda NT i ett realtidssystem så är det ett antal frågor som måste besvaras, några intressanta är:

- Vad skiljer NT från kommersiella RTOS?
- Är de grundläggande mekanismerna i NT implementerade på ett sådant sätt att determinism kan uppnås?
- Om man inte kan använda NT för hårda realtidssystem, vilken typ av mjuka realtidssystem kan man då använda NT till? Kan man finna någon typ av klass av problem som är möjliga att implementera på NT? Till exempel applikationer som inte gör någon diskaccess och som ska exekveras med 100Hz där man tillåter en försening med upp till 10ms.
- Hur mycket funktionalitet kan man stänga av i NT och fortfarande ha ett system som är användbart? Påverkar detta realtidsprestandan?
- Hur hanterar NT avbrott från yttre enheter? Dvs hur är drivrutinerna mot hårdvaran uppbyggda?

Många företag marknadsför speciella utökningar och drivrutiner till NT som ska ge realtidsegenskaper så man kan på förhand säga att NTs realtidsegenskaper an-

tagligen inte är de bästa och att en marknad verkar finnas för realtidssystem under NT. Dessa utökningar är i detta fall inte intressanta utan man vill veta vad man kan göra med helt vanlig “direkt ur kartongen” NT.

1.2 Syfte

Syftet med examensarbetet är att undersöka Windows NT ur ett realtidsperspektiv. Fokus ligger speciellt på hur Microsoft valt att implementera de grundläggande tjänsterna i NTs kärna.

1.3 Problemformulering

Här formuleras den problemställning som behandlas i examensarbetet.

1.3.1 Problemdefinition

Kommersiella operativsystem avsedda för realtidstillämpningar är specialiserade system, noggrant designade med fokus på en sak, nämligen realtidsegenskaper. Windows NT å andra sidan är ett generellt operativsystem avsett för skrivbordet eller mindre servrar i en nätverksmiljö.

När man utvecklar realtidssystem så finns ett antal mekanismer i operativsystemet som är önskvärda för att ge möjlighet att använda det som värd för ett realtidssystem. Frågan är om motsvarigheter till dessa finns på NT-plattformen och om de ej finns, hur kan man implementera dessa?

Om kärnan i NT är designad på ett sådant sätt att determinism inte kan uppnås så kan inte heller något system som använder sig av kärnans tjänster bli deterministiskt. En undersökning av hur de grundläggande mekanismerna i NT är designade med avseende på realtidsprestanda ger svaret på om det är möjligt att utveckla deterministiska system baserade på NT.

1.3.2 Önskade resultat

- En beskrivning av några av de vanligaste kommersiella RTOSen.
- Identifiering av de mekanismer som kan förväntas finnas i ett RTOS.
- Beskrivning av motsvarande mekanismer i NT.
- Undersökning av hur de tjänster i operativsystemet som används är implementerade med fokus på realtidsegenskaper.
- Praktiska tester av tjänsterna i operativsystemet för att få fram mätdata på de mekanismer som identifierats.

2 Bakgrund RTOS

I detta kapitel definieras vad som menas med realtid och realtidssystem. Här definieras även vissa grundläggande begrepp som ska göra det möjligt för oss att diskutera egenskaper och struktur hos operativsystem.

2.1 Vad är ett realtidsoperativsystem / realtidssystem?

Att definiera vad ett realtidssystem eller ett realtidsoperativsystem har för egenskaper är inte lätt då det finns ett flertal olika definitioner på dessa termer. Det räcker med att följa news-gruppen `comp.os.realtime` i någon vecka så kommer man garanterat att ha sett i alla fall ett par heta diskussioner rörande definitionerna på dessa termer.

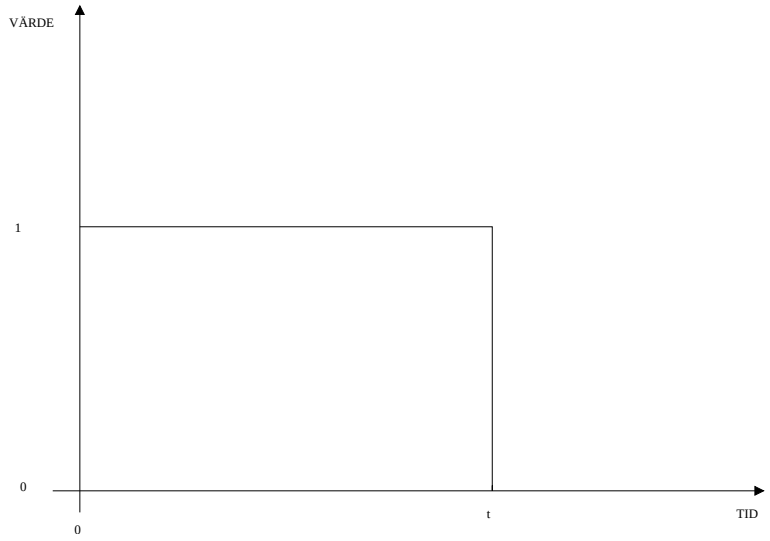
Den definition av realtidssystem och realtidsoperativsystem som presenteras nedan följer den i [Krishna & Shin 97].

Ett realtidssystem är ett system där tidskrav ställs på vissa operationer. Det kan tex vara att systemet måste hämta ett visst mätvärde var 15 ms annars skrivs det över. För att enklare kunna illustrera detta kan man införa konceptet med värderingsfunktioner för operationer. En värderingsfunktion är en funktion av tiden som ger värdet av en viss operation vid en viss tidpunkt. Beroende på vilken typ av operation det rör sig om så ser värderingsfunktionen olika ut.

Figur 1 - Värderingsfunktion för funktionsanrop i ett icke realtids system

I ett icke realtidssystem kan värdet på ett funktionsanrop se ut som ovan. Värdet av funktionsanropet beror inte av tiden. Användaren blir antagligen irriterad om det tar lång tid men programmet löser sin uppgift oavsett hur lång tid användaren fått vänta.

Realtidssystem har en värderingsfunktion som abrupt går till noll vid en viss tidpunkt. Detta betyder att värdet av en försenad operation är noll, det är lika illa att försöka och misslyckas med att möta tidskraven som att inte försöka överhuvudtaget.

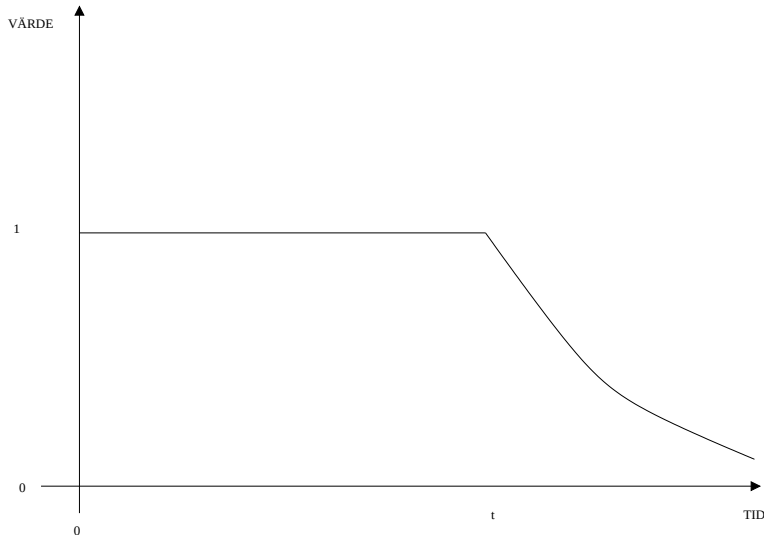


Figur 2 - Värderingsfunktion för funktionsanrop i hårt realtidssystem

I figuren ovan går anropets värde till 0 vid tidpunkten *t* det vill säga anropet har inget värde för oss efter tidpunkten *t*.

Denna definition definierar den typ av system som brukar kallas hårda realtidssystem. I system av denna typ är det ofta förenat med stora kostnader eller risker att missa en deadline. Exempel på hårda realtidssystem är tex styrsystem för flygplan, system för processstyrning och medicinsk utrustning såsom röntgenapparater.

Om vi har ett system där det inte är lika farligt att missa en deadline utan det är något som kan få hända men väldigt sällan så har vi ett sk mjukt realtidssystem.



Figur 3 - Värderingsfunktion för funktionsanrop i mjukt realtidssystem

I figuren ovan ses ett exempel på en värderingsfunktion för en funktion i ett mjukt realtidssystem. Efter tidpunkten *t* avtar värdet på funktionen. Detta kan till exempel ses som värdet på ett datapaket vid IP-telefoni efter en viss tidpunkt är paketet inte särskilt intressant längre då det hela tiden kommer fler paket som är färskare och därför viktigare.

Exempel på mjuka realtidssystem är IP-telefoni, videotelefoni, VR-system och flygsimulatorer.

Ett realtidssystem har alltså ett predikterbart beteende. För att det ska vara möjligt att bygga ett sådant system måste vi ha ett operativsystem som erbjuder tjänster som har predikterbart beteende. Ett operativsystem som erbjuder detta kallas för realtidsoperativsystem. Det är viktigt att notera att ett realtidssystem kräver ett realtidsoperativsystem men det finns inget som garanterar att ett system byggt på ett realtidsoperativsystem blir ett realtidssystem.

2.2 Definitioner

RTS: Realtidssystem se avsnitt 2.1: "Vad är ett realtidsoperativsystem / realtidssystem?".

RTOS: Realtidsoperativsystem se avsnitt 2.1: "Vad är ett realtidsoperativsystem / realtidssystem?".

Process: Ur systemets perspektiv så är en process den minsta exekverande enhet som självständigt kan begära systemresurser. En process lever i en virtuell värld isolerad från andra processer, denna miljö tillhandahålls av operativsystemet.

Konceptuellt så kan en process tänkas exekvera samtidigt och oberoende med andra processer. Operativsystemet byter automatiskt mellan olika processer som är färdiga att köra. Dessa processer ligger lagrade i en kö. En process hamnar i

kön tex vid systemanrop, ett anrop kan få en process att avbrytas och en annan att fortsätta sin exekvering.

Även om varje process är en självständigt exekverande enhet så måste den koordinera sig med andra processer eller interupthanteringsrutiner med hjälp av de synkroniseringstjänster som operativsystemet erbjuder.

Tråd: En process kan bestå av en eller flera trådar. Till skillnad från processer så delar samtliga trådar i en process på ickelokala variabler, filhandtag och andra delade resurser. Varje tråd har en unik programräknare och en egen stack för övrigt delas all data mellan trådarna.

Processorutnyttjande: Låt e_i motsvara den tid det tar att exekvera process *i*. Låt P_i vara periodtiden hos process *i*.
Processens processorutnyttjande ges då av $\frac{e_i}{P_i}$.

Totalt processorutnyttjande: Om antalet processer i systemet är n ges det totalt processorutnytjandet av

$$U = \sum_{i=1}^{n} \frac{e_i}{P_i}$$ *Ekvation 1 - Totalt processorutnyttjande*

ISR: Interupt Service Routine, den rutin som anropas av operativsystemet när ett interupt sker.

IPC: Inter Process Communication, kommunikation mellan olika processer i systemet. Sker oftast via meddelanden eller händelser men kan även ske t ex via pipes.

2.3 Rate Monotonic Scheduling

I ett realtids system har man ett antal olika processer och trådar som har speciella krav på sig vad gäller periodicitet, prioritet och så vidare. Att schemalägga dessa så att kraven uppfylls är svårt och ännu svårare är att visa att en viss schemaläggning uppfyller de krav som ställts. För att hantera detta har en rad olika algoritmer utvecklats. En av de äldsta är den så kallade *rate-monotonic scheduling algoritmen* (RMS). Följande beskrivning av RMS är baserad på [Krishna & Shin 97].

För att RMS ska ge giltiga resultat måste följande krav vara uppfyllda

- Alla processer kan avbrytas var som helst och kostnaden för att avbryta den är försumbar. Detta betyder att koden inte får innehålla några critical-sections eller andra delar av koden där processen ej kan avbrytas.
- Den enda begränsande resursen är processortid. Minne I/O och andra resurskrav kan försummas.
- Alla processer är oberoende. Inga krav på turordning existerar.
- Alla processer är periodiska.

- En process relativa deadline är samma som dess period. Det vill säga endast en process finns endast i ett exemplar.

Det finns ett enkelt test för att avgöra om en mängd processer är möjliga att schemalägga med RMS nämligen

$$U \leq n\left(2^{\frac{1}{n}} - 1\right)$$

Ekvation 2 - Villkor för schemaläggning med RMS

där *n* är antalet processer som ska schemaläggas. Detta är ett tillräckligt men inte nödvändigt krav, dvs det finns processmängder som är möjliga att schemalägga som inte uppfyller kravet.

En av de centrala punkterna i algoritmen är att man tilldelar varje process en egen prioritetsnivå. Den process som har den lägsta periodtiden tilldelas den högsta prioriteten. Den process som har näst lägst periodtid tilldelas nästa prioritetsnivå och så vidare.

Detta är bara en kort introduktion till algoritmen och om processmängden uppfyller de krav som presenterats så räcker detta, mer om algoritmen kan man finna i [Krishna & Shin 97].

2.4 Priority Inversion och Priority Inheritance

I ett system finns normalt förutom trådar med olika prioriteter ofta någon typ av gemensamma resurser ofta i form av hårdvara. Operationer på dessa resurser är ofta icke-atomiska. Man är då tvungen att använda en critical-section som skydd för den del av tråden som använder den delade resursen. När man använder critical-sections i ett prioritetsbaserat system kan man få ett fenomen som kallas *priority-inversion* vilket är när en tråd med lägre prioritet hindrar en tråd med högra prioritet att exekvera.

Priority inversion illustreras bäst med ett exempel. Om vi har ett tänkt system med 3 processer:

P1, vilket är en process med hög prioritet och som utnyttjar en delad resurs skyddad med hjälp av C1.

P2, är en process med medelhög prioritet

P3, är en process med låg prioritet och som utnyttjar en delad resurs skyddad med hjälp av C1.

C1, är en critical-section

Antag nu att vi har följande scenario.

1. Process P3 har exekverat ett tag och kommit in i C1.
2. Process P1 blir nu klar för exekvering och avbryter P3
3. P1 exekverar tills den vill in i den critical-section som skyddas av C1. P1 lägger sig nu och väntar på att få komma in i C1.
4. P3 exekverar vidare.
5. Process P2 blir nu klar för exekvering och avbryter P3.

6. Process P1 hindras från att exekvera av P3 som i sin tur har lägre prioritet än P2 och får alltså inte exekvera.

Det som nu har hänt är att en process har hindrat en process på en högre prioritetsnivå från att få exekvera.

Det centrala i problemet var den critical section som delades av P1 och P3, hur ska man då hantera delade resurser för att undvika priority inversion? Lösningen på detta är *priority inheritance* vilket innebär att den process som befinner sig i en critical section temporärt ärver den högsta prioritetsnivån bland de andra processer som väntar på denna critical-section. Det tidigare scenariot övergår då i följande.

1. Process P3 har exekverat ett tag och kommit in i C1.
2. Process P1 blir nu klar för exekvering och avbryter P3
3. P1 exekverar tills den vill in i den critical-section som skyddas av C1. P1 lägger sig nu och väntar på att få komma in i C1. Prioriteten för P3 höjs nu till samma nivå som P1
4. P3 exekverar vidare.
5. Process P2 blir nu klar för exekvering men har lägre prioritet än P3.
6. Process P3 lämnar C1

I detta fall hanterades problemet med delade critical sections på ett sätt som ledde till att priority inversion undveks.

3 Beskrivning pSOS

I detta kapitel beskrivs operativsystemet pSOS från Integrated Systems Inc. När olika mekanismer beskrivs eller när termen “operativsystemet” används så avser det här pSOS. Kapitlet är baserat på [ISI98].

3.1 Processhantering

I detta stycke presenteras hur pSOS hanterar processer.

3.1.1 En process olika tillstånd

En process kan befinna sig i flera olika tillstånd, ett byte av tillstånd kan bara ske i samband med ett anrop till operativsystemet antingen av processen självt, en annan process eller en ISR.

Ur operativsystemets perspektiv så existerar inte en process innan den är skapad eller efter dess avslutande. En process måste startas innan den kan exekvera. Innan en skapad process startas är den någon form av embryo process.

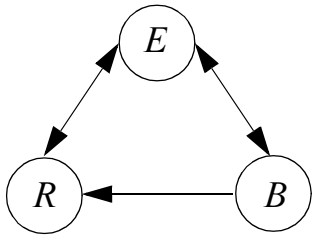
När processen väl har startats så kan den befinna sig i något av tre tillstånd:

- Redo (Ready)
- Exekverande (Running)
- Blockerad (Blocked)

En process som *redo* är körbar och ickeblockerad den väntar endast på att processer med högre prioritet ska lämna ifrån sig processorn. Då en process endast kan startas genom ett anrop från en annan exekverande process så föds alltid processer redo.

En *exekverande* process är en redo process som har fått tillgång till processorn. Det existerar alltid endast en exekverande process. I allmänhet så har den exekverande processen den högsta prioriteten av alla *redo* processer (om inte preemption stängts av).

En process blockeras endast som ett resultat av en avsiktlig handling utförd av processen självt, i allmänhet så har processen utfört ett systemanrop som innebär att den anropande processen väntar på ett resultat. En process kan därför inte gå från tillståndet *redo* till *blockerad* då endast exekverande processer kan utföra systemanrop. I figur 4 så visas de möjliga tillståndsövergångarna.



Figur 4 - Tillståndsövergångar för processer.
E=Exekverande, R=Redo, B=Blockerad

(E->B) En exekverande process kan blockeras tex då den hämtar ett meddelande från en tom meddelandekö eller då den väntar på en semafor.

(**B->R)** En blockerad process (B) blir *Redo* tex då ett meddelande anländer till en meddelandekö där B väntat, en semafor återlämnas och B är först i kön av väntande processer.

(B->E) En blockerad process (B) blir exekverande då en övergång **B->R** sker och B har högre prioritet än den senast exekverande processen.

(R->E) En *Redo* process (R) blir exekverande tex då den senast exekverande processen (E) blir blockerad eller då E ändrar sin eller R:s prioritet så att R får högre prioritet än E och ingen annan process i systemet har högre prioritet än R.

(E->R) Den exekverande processen (E) blir redo när en övergång B->E sker för en blockerad process eller en övergång R-> sker får en annan process.

3.1.2 Schemaläggning

Systemkärnan i pSOS använder en prioritetsbaserad algoritm för avbrytande (preemptive)processbyten. Det vill säga vid varje tidpunkt så körs den process som har högst prioritet av alla körbara processer. Man har dock möjlighet att förbjuda avbrytande samt införa timeslicing och därigenom påverka schemaläggarens uppträdande.

Varje process har ett *kontroll-ord* som innehåller två bitar som påverkar schemaläggningen. En bit kontrollerar om processen kan avbrytas av en annan process eller ej. Om preemption är avstängd så kan ingen annan process avbryta denna när den väl börjat exekvera, inte ens processer med högre prioritet kan avbryta den. En task-switch kommer då att äga rum när processen blockeras på något anrop eller om den tillåter preemption igen.

Den andra biten kontrollerar timeslicing. Om processen har timeslicing aktiverat så håller operativsystemet reda på hur länge processen kört, när processen exekverat mer än en förutbestämd tid så avbryts den och nästa process med samma prioritet får exekvera (om det finns några sådana). Schemaläggningen sker då via en roundrobin algoritm där storleken på en timeslice kan väljas fritt.

Varje process har en timeslice-räknare som räknas ned av kärnan efter hand och när räknaren når noll så sker ett av två möjliga fall

1. Om alla andra processer har lägre prioritet så sker ingenting, timeslice-räknaren står kvar på noll och processen exekverar vidare.
2. Om en eller flera processer med samma prioritet är färdiga för att köras så byter den exekverande processen tillstånd till *ready,* dess timeslice-räknare sätts till sitt startvärde och processen placeras sist i kön av väntande processer med samma prioritet. Den process som ligger först i kön exekveras nu.

Oavsett om roundrobin-schemaläggning är aktiverat eller ej så placeras en process som byter från blockerad till redo i slutet av kön av processer med samma prioritet. På samma gång så sätts dess timeslice-räknare till sitt ursprungsvärde.

3.1.3 Prioritetsnivåer

Varje process som skapas måste ha en prioritetsnivå tilldelad sig. Det finns 256 olika prioritetsnivåer, 0 är lägst och 255 högst. Nivåerna 0, 240 - 255 är reserverade av operativsystemet. En process prioritet kan ändras under körning genom systemanrop.

När en process är redo för att exekveras så placerar operativsystemet processen i en indexerad kö av färdiga processer. Den placeras efter alla processer med lika eller högre prioritet. Alla köoperationer inklusive insättning och borttagning sker snabbt och i konstant tid.

3.2 Minneshantering

I detta stycke beskrivs hur pSOS hanterar RAM-minne.

3.2.1 Minnestyper

I pSOS så finns två typer av minne:

- **Regioner**: En region är ett fast område i minnet som har en startadress, och en längd. En region kan ligga i ett lokalt RAM-minne eller i ett delat minne som hela systemet kan komma åt. Flera olika regioner kan inte använda samma minnesområde.
- **Segment**: Ett segment är ett minnesområde i en region allokerad av operativsystemet. När man allokerar ett segment så anger man endast hur mycket minne man behöver.

Regioner används i allmänhet när man vill använda minnesmappade enheter då man kan specificera var i adressrymden minnet ska allokeras. Segment används när man bara behöver temporärt minne för någon operation eller om man bara behöver minne och det är likgiltigt var det finns i adressrymden.

3.2.2 Allokeringsalgoritm

Varje region har en RNCB (Region Control Block) som innehåller information om vilka delar av regionen som är allokerad och en standardstorlek på de element som allokeras. Denna storlek måste vara av typen $\{2^n, n \in Z, n > 5\}$ Varje segment som allokeras får en storlek som är närmaste multipel av storleken. Dvs om storleken är 32 byte och en begäran om 130 bytes görs så allokeras 5 enheter eller 160 bytes.

Operativsystemet lagrar de tillgängliga minnesblocken i en heapstruktur, en regions RNCB innehåller en allokeringstabell och heapstrukturen. Dom fria segmenten lagras sorterade i avtagande storlek så när en allokering ska göras så behöver bara storleken på det första segmentet kontrolleras för att avgöra om allokeringen kan utföras eller ej. Om det första segmentet är för litet så finns inget sammanhängande segment som är stort nog. Om det finns ett segment som är stort nog så delas det, en del returneras till den anropande processen och den andra delen sätts åter in i heapstrukturen, om det fria blocket exakt motsvarar den efterfrågade storleken så sker ingen delning. När ett segment avallokeras så för-

söker man alltid sätta ihop det med lediga intillliggande segment för att få ett så stort sammanhängande område som möjligt.

3.3 Meddelandeköer

De systemanrop som används för att hantera köer säger väldigt mycket om kösystemet

```
q_create     Skapa en meddelandekö
q_ident      Returnera en kö:s ID
q_delete
q_recieve    Hämta ett meddelande från kön, möjlighet
             finns att vänta tills ett meddelande
             finns tillgängligt
q_send       Sätta in ett meddelande i slutet av kön
q_urgent     Sätta in ett meddelande först i kön
q_broadcast  Sänder ett meddelande till samtliga
             processer som väntar på en kö
```

När en kö skapas så har man möjlighet att välja om processer ska få tillgång till kön enligt FIFO-ordning eller prioritetsordning. Man har även möjlighet att begränsa köns längd. En kö är inte bunden till en viss process utan en eller flera processer kan sända meddelanden till en kö och en eller flera processer kan hämta meddelanden ur kön. En meddelandekö är alltså en många till många förbindelse.

Det finns två typer av köer i systemet, en kö med en meddelandestorlek på 16 byte och en med valfri storlek på meddelandena.

3.4 Händelser och asynkrona signaler

Detta stycke beskriver pSOS händelser och meddelanden.

3.4.1 Händelser

En process har möjlighet att kommunicera via händelser. Varje process har 32 händelseflaggor som är möjliga att vänta på. Dessa flaggor är bitvis kodade som ett 32bitars ord. De översta 16bitarna är reserverade för systemet men de 16 nedersta är tillgängliga för användaren. Händelser har ett väldigt enkelt gränssnitt, två anrop finns `ev_receive` och `ev_send`.

`Ev_send` används för att skicka en eller flera händelser till en annan process. Med `ev_receieve` så kan en process vänta med eller utan timeout på en händelse, en av flera händelser (OR) eller samtliga av flera händelser (AND).

Om en process väntar på en eller flera händelser och de redan har inträffat så rensas bitarna och anropet returnerar genast.

3.4.2 Skillnaden mellan händelser och meddelanden

- En händelse kan användas för att synkronisera en process men den innehåller ingen ytterligare information.

- En händelse kan bara skickas till en specifik process dvs sändaren måste känna till processen händelsen skickas till. När en process vill skicka ett meddelande så behöver den endast känna till kön, den vet inte vilka eller hur många processer som hämtar meddelanden ur kön.
- En process kan samtidigt vänta på flera händelser men endast på ett meddelande.
- Meddelanden buffras och köas automatiskt. Händelser räknas eller köas ej, om en händelsebit är satt och en likadan händelse inträffar så påverkas ej händelsebiten.

3.4.3 Asynkrona signaler

En process kan även ha en ASR (Asynchronous Service Routine). Syftet är att ge en process två exekveringsslingor, en som normalt exekveras och en som exekveras när en asynkron signal kommer in. Med hjälp av signaler så kan en annan process tvinga en process att exekvera sin ASR. Signaler är alltså en typ av mjukvaruinterupt. Varje process har 32 olika signaler kodade i ett 32 bitars ord. De 16 översta bitarna är reserverade för systemet men de 16 nedersta är tillgängliga för användaren. Oavsett vilken signal som skickas så anropas samma ASR.

När en signal skickas så exekveras inte ASRen omgående utan nästa gång målprocesser får köra så exekveras den. Asynkrona signaler har samma egenskaper som händelser dvs oavsett hur många gånger en signal skickats så syns den endast en gång. En process har möjlighet att stänga av ASR-hanteringen under exekveringen. Man kan då vara säker på att vissa delar i programmet inte avbryts.

3.4.4 Skillnaden mellan signaler och händelser

För att behandla en händelse så måste målprocessen explicit vänta på den, en signal å andra sidan medför att ASRen exekveras omgående om den existerar och inte stängts av

3.5 Synkronisering

pSOS använder sig av traditionella semaforer för synkronisering. När en semafor skapas så har man möjlighet att bestämma hur många *tokens* som ska finnas och om väntande processer ska behandlas i FIFO eller prioritetsordning. Kön med väntande processer är implementerad som en dubbellänkad lista.

Semaforerna stödjer de klassiska P & V primitiverna. När en process efterfrågar en *token* med sm_p så minskas *token*-räknaren med ett om den är skild från noll och anropet returnerar. Om räknaren är noll så kan den anropande välja mellan att vänta, vänta med timeout eller returnera med felmeddelande.

Sm_v ökar *token*-räknarens värde med 1. Om några processer finns i kön med väntande så får den första processen en *token* och flyttas till tillståndet ready.

3.6 Tidshantering

pSOS ger möjlighet till synkronisering med omvärlden genom två olika mekanismer. Man kan dels blockera sin process en viss tidsrymd eller fram till en spe-

ciell tidpunkt och man kan få en händelse skickad till sig efter en viss tid eller vid en speciell tidpunkt.

Systemets klocka stegas upp med hjälp av ett systemanrop tm_tick som normalt sker från en realtidsklockas ISR vid varje timeravbrott. Frekvensen på tm_tick anropen bestämmer upplösningen på systemets klocka.

Operativsystemet innehåller en kalender med datum och tid vilken kan användas för att vänta på en viss tidpunkt.

Dessa funktioner används även av systemet för att avgöra när timeouten på vissa systemanrop nåtts. Timeout-tiden mäts i antal tick så om frekvensen på klockan är 100Hz och man vill ha ett timeoutintervall på 50ms så ska ett värde på 5 anges. Timeout värdet sparas i ett 32bitars ord så det finns en övre gräns hur lång timeout man kan ha. Ett timeout värde på n kommer att gå ut på det n-te ticket efter anropet. Då anropet kan ske när som helst mellan två ticks så kommer den verkliga väntetiden att vara mellan n-1 och n ticks.

3.7 Avbrottshantering

I realtidssystem så ingår ofta avbrott som en central del av systemet. Den rutin i pSOS som hanterar avbrott kallas ISR (Interupt Service Routine). När ett avbrott kommer så avbryts alla andra processer och avbrottets ISR anropas, en ISR kan endast avbrytas av andra avbrott med högre prioritet. Under avbrottsbehandlingen så sätter man alltså den normala schemaläggningsalgoritmen ur spel. Det är därför väldigt viktigt att man minimerar storleken på sin ISR.

Då ISRen ska exekveras så fort som möjligt så kan man inte göra några systemanrop som medför att man blir blockerad. En ISR kan därför endast använda vissa anrop som kan ses som meddelandekällor tex höja värdet på en semafors tokenräknare, skicka ett meddelande eller en händelse. Man kan inte heller använda synkroniseringsobjekt som inte är lokala dvs ligger lokalt på den maskin man kör på.

Ett typiskt fall av avbrottshantering kan tänkas se ut som följer. En kö används för kommunikation mellan en ISR och en process. Processen frågar efter ett meddelande ur kön och väntar på att det ska komma. Processen blockeras i väntan på ett meddelande. När ett avbrott sker så skickar ISRen ett meddelande till kön och returnerar. När ISRen returnerar så startar den normala schemaläggningen processen som väntar på meddelandet flyttas till tillståndet redo och körs nästa gång den är den körbara process som har högst prioritet.

4 OSE

Detta kapitel beskriver översiktligt realtidsoperativsystemet OSE från ENEA Data. Kapitlet är huvudsakligen baserat på [OSE98].

4.1 Processhantering

I detta stycke beskrivs hur OSE behandlar processer.

4.1.1 Typer

I OSE finns ett antal olika typer av processer.

- **Interupt processer**: Dessa anropas när ett hårdvaruavbrott eller en mjukvaru-händelse genererats och kommer att exekvera från början till slut utan avbrott om inte ett avbrott med högre prioritet kommer.
- **Timer interupt processer**: Dessa uppförs sig på exakt samma sätt som en interupt-process förutom att de anropas som en följd av att system-timern har nått en viss tidpunkt.
- **Prioritized processer**: Detta är den vanligaste processtypen. Dessa processer är skrivna som oändliga loopar som körs runt, så länge inga avbrott sker eller någon annan prioritized-process med högre prioritet är redo att köras.
- **Background processer:** Dessa körs enligt en strikt time-sharing schemaläggning på en prioritet under prioritized-processes. Även dessa är skrivna som oändliga loopar på samma sätt som prioritized-processer. Syftet med denna typ är att dela överbliven cpu-tid mellan ett antal processer. När en process startats så kör den hela sin timeslice utan att avbrytas av andra background-processes. Det vill säga att om processen väntar på något och hinner vakna igen innan dess timeslice är slut så kommer den att fortsätta exekvera. När en process kört slut på sin timelice så placeras den sist i kön igen och nästa process börjar köra.
- **Phantom processer:** En fantom-process är en process som inte innehåller någon kod. Den kan därför inte ta emot några signaler. Om en signal skickas till en fantom-process och det inte finns någon redirection-table så kommer signalen bara att försvinna och dess minnesutrymme rensas. Fantomprocesser används främst för att via en redirection-tabell användas som *router* och skicka meddelanden vidare till andra processer.

4.1.2 Schemaläggning

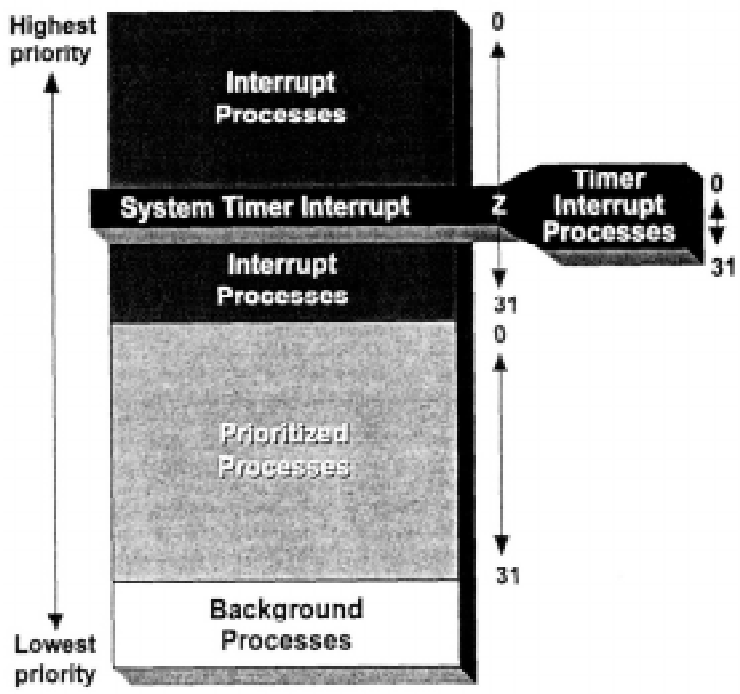
I OSE så finns 4 typer av schemaläggning

- **Pre-emptive**: OS kan när som helst avbryta processen, även om den utför ett anrop till operativsystemet. Det vill säga operativsystemet kan avbryta en process och låta en annan exekvera när som helst. Alla processer i OSE använder denna metod.
- **Cyclic**: Processer kan schemaläggas för körning med en viss frekvens. Denna schemaläggningsmetod används för timer-interupt processer.

- **Priority Based**: Den process med den högsta prioriteten kommer att köras så länge inga interupt kommer och processen inte väntar på någon händelse. Denna schemläggningsmetod används för alla prioritized processer i OSE.
- **Round Robin**: För att alla processer på en viss prioritetsnivå ska få tillgång till lika mycket CPU-tid så har varje prioritetsnivå en kö som innehåller alla processer som är redo att köras. Processen som ligger först i kön är den som för tillfället körs. När en process förlorar rätten till processorn så läggs den sist i kön och får inte exekvera innan den har hamnat först i kön igen. Både prioritized och background processer in OSE använder denna schemaläggningsprincip. En skillnad mellan dem är dock att en priotitized process placeras i slutet av kön så fort den väntar på något, en background process däremot körs tills dess timeslice är slut.

4.1.3 Prioritet

Varje process har en prioritetsnivå mellan 0 och 31 där 0 är den högsta prioriteten. Prioritetsnivån har dock en speciell betydelse för varje klass av processer och de olika klasserna har olika inbördes prioriteter.



*Figur 5 - Prioritetsnivåer i OSE**[OSE98]***

- **Interupt processer:** Har en prioritet som överensstämmer med den prioritet motsvarande hårdvaruavbrott har.

- **Timer Interupt Processer:** Dessa har en prioritet som indikerar i vilken ordning olika processer som ska köras vid exakt samma tidpunkt startas.
- **Prioritized Processer:** Här beskriver prioritetsvärdet i vilken ordning olika processer ska köras när någon process blockeras och vilka processer som avbryter varandra.
- **Background Processer:** För dessa används inte prioritetsvärdet.

4.2 Minneshantering

Centralt i OSEs minneshantering är begreppet *pool*. En pool är ett minnesområde som ett antal processer allokerar buffertar, stack, signaler med mera ur. I systemet finns alltid en global pool, systempoolen. Man kan även skapa lokala pooler i minnesutrymmet som tillhör de processer poolen tillhör.

Systempoolen är den första som skapas, den används av kärnan och måste därför vara åtkomlig för den. Man kan göra ett system där alla processer allokerar minne i systempoolen, ett fel i någon process kan dock enkelt krascha kärnan.

Man har då möjlighet att gruppera en eller flera processer i en enhet som kallas block och sedan låta detta block ha en egen pool. Om informationen i poolen blir förstörd så påverkar detta endast processerna i det block som ägde poolen.

Normalt används pekare för att skicka t ex meddelanden mellan olika processer dvs en pekare till en adress i sändarens pool skickas till mottagaren. Detta medför att mottagaren kan förstöra data i sändarens pool. För att undvika detta så har man infört ännu en gruppering, *segment.* Om ett meddelande skickas mellan två processer som tillhör pooler i olika segment så kan man välja om man vill skicka en pekare eller om man vill kopiera meddelandet.

För att få fullt minnesskydd mellan olika processer så bör poolerna placeras i olika segment som isoleras med hjälp av en MMU (Memory Management Unit).

4.3 Meddelandeköer

OSE använder begreppet signaler som beteckning för meddelanden mellan processer. För att kunna skicka en signal till en process så måste man känna till processens ID. När man ska skicka en signal så måste man allokera minne för den i den pool man tillhör. Varje signal har ett signal nummer som identifierar vilken typ av signal det är. Processer har möjlighet att specificera vilka signaltyper de är intresserade av.

Den mottagande processen kan antingen vänta på en viss signal eller polla efter den. Mottagaren kan också välja vilken signal den vill hämta ur kön för behandling. Signaler som filtrerats bort eller som inte är intressanta just nu ligger kvar i signalkön för senare behandling.

En unik mekanism hos OSE är möjligheten till *Signal Redirection* varje process kan ha en tabell som innehåller signalnummer och process-id. Om en signal kommer in av en typ som finns med i tabellen så skickas signalen vidare till den process vars process-id angetts i tabellen. En signal kan passera flera redirection tables innan den kommer fram till den slutgiltiga mottagaren. Kärnan har funk-

tionalitet för att upptäcka loopar i tabellerna och ignorerar då tabellen och skickar meddelandet till den första processen i loopen.

Denna mekanism används ofta för att skapa ett stabilt gränssnitt in i en modul, man skapar en fantomprocess som har en redirection-tabell till andra processer i modulen. Man kan då ändra struktur och ansvarsområden för processerna inuti modulen utan att behöva ändra gränssnittet utåt.

4.4 Synkronisering

För synkronisering mellan processer så används semaforer. I systemet finns två olika sorters semaforer vanliga och snabba.

En vanlig semafor ägs inte av någon unik process utan vilken bakgrunds eller prioritized process som helst kan använda den. Varje process kan ha flera semaforer för olika uppgifter.

Varje process har en och endast en snabb semafor. Den enda process som kan vänta på en snabb semafor är den som äger den.

Precis som namnet säger är en snabb semafor snabbare än en vanlig.

4.5 Tidshantering

I OSE är timeravbrott den mekanism som ger möjlighet att koppla tidsintervall till exekvering av processer. Vid varje tick så kontrollerar kärnan om något eller flera timeravbrott ska startas om så är fallet startas den med högst prioritet, därefter nästa osv. För att något timeravbrott inte ska svälta ut något annat eller leda till att systemet missar ett tick så måste summan av exekveringstiden för alla timeravbrott vara mindre än tiden mellan två tick.

Om man använder en separat realtidsklocks-modul finns möjlighet till att sätta alarm till vissa tidpunkter och datum samt funktioner för översättning mellan olika representationer av tid.

4.6 Avbrottshantering

Avbrott i OSE hanteras på i stort sätt samma sätt som i andra operativsystem. När ett hårdvaruavbrott sker så anropas en viss rutin för att hantera detta. Denna rutin kan bara avbrytas av andra avbrott med högre prioritet.

Avbrottshanteringsrutinen kan inte heller blockera på något utan får bara utföra operationer som inte blockerar.

Något som skiljer OSE från andra operativsystem är möjligheten att enkelt orsaka ett mjukvaruavbrott genom att skicka en signal till avbrottsprocessen eller genom att signalera dess snabba semafor. Den väckta avbrottsprocessen kan se skillnad mellan ett riktigt hårdvaruavbrott och ett mjukvaruavbrott. Man kan till exempel tänka sig att man använder denna mekanism för kommunikation med någon yttre enhet, en process lägger upp ett meddelande i en buffert och väcker interuptrutinen med ett mjukvaruavbrott, varefter hårdvaran skickar varje byte så kommer ett hårdvaruavbrott med begäran om fler bytes som ska skickas.

5 Bakgrund NT

Detta kapitel ger en inblick i de mål som Microsoft hade med Windows NT när utvecklingsarbetet påbörjades. Kapitlet innehåller även en beskrivning av NT-s arkitektur.

5.1 Historik / Designmål NT

När specifikationerna på NT gjordes 1989 så hade man följande krav

- Operativsystemet ska vara ett 32-bitars operativsystem med preemptive multitasking och virtuellt minne.
- Operativsystemet ska i så stor omfattning som möjligt vara reentrant. Det vill säga själva operativsystemet ska vara trådat.
- Det ska finnas på flera plattformar och arkitekturer
- Det ska kunna köras på flerprocessorsystem och utnyttja processorerna effektivt.
- Bli en bra plattform för *distributed computing* både som server och klient.
- De flesta 16-bitars MS-DOS och Windows 3.1 applikationerna ska kunna köras.
- Möta den amerikanska regeringens och industrins säkerhetskrav.
- Uppfylla POSIX 1003.1 standarden enligt krav från den amerikanska regeringen.
- Stödja UNICODE så att systemet på ett enkelt sätt kan anpassas till andra språk och teckenuppsättningar.

För att ge ett stöd i alla de designbeslut som skulle tas så satte man även upp följande designmål [Solomon 98].

- ***Extensibility****. The code must be written to comfortably grow and change as market requirements change.*
- ***Portability****. The system must be able to run on multiple hardware architectures and must be able to move with relative ease to new ones as market dictates.*
- ***Reliability*** *and robustness. The system should protect itself from both internal malfunction and external tampering. Applications should not be able to harm the operating system or other running applications.*
- ***Compatibility****. Although Windows NT should extend existing technology, its user interface and older application programming interfaces (APIs) should be compatible with older versions of Windows as well as older operating systems such as MS-DOS. It should also interoperate well with other systems such as UNIX, OS/2, and NetWare.*
- ***Performance****. Within the constraints of the other design goals the system should be as fast and responsive as possible in each hardware platform.*

Dessa krav och mål är bra att ha i bakhuvudet när man studerar NTs struktur och uppbyggnad.

5.2 Rättigheter och prioriteter inom NT

Ett operativsystem använder sig ofta av någon form av minnesskydd och olika rättigheter för operativsystem och applikationer i systemet. Detta för att ett program inte ska kunna påverka operativsystemets stabilitet.

I NT exekverar ett program i något av två olika moder, *kernel-mode* eller *user-mode* dessa har olika rättigheter. Användarens applikationer exekverar i user-mode och operativsystemet i kernel-mode. En process som exekverar i user-mode har inte direkt tillgång till hårdvaran och vissa maskininstruktioner, program i kernelmode däremot har fri tillgång till hela systemet. En applikation i user-mode som kraschar kan alltså inte förstöra mer än sin egen minnesrymd.

x86 processorn definierar fyra nivåer av rättigheter, *ringar* för att skydda systemets kod och data från applikationer med mindre rättigheter. NT använder ring 0 för kernel-mode och ring 3 för user-mode. Anledningen till att NT bara använder två nivåer är att systemet ska vara portbart till andra arkitekturer. De RISC-baserade arkitekturerna har oftast bara två nivåer av rättigheter.

Man har även delat processorns adressrymd i två halvor, de första 4 GB från $00000000 till $7fffffff kan användas av processer i både user och kernel mode minnet mellan $80000000 och $ffffffff kan bara användas av processer i kernelmode.

NT har ingen skyddsmekanism för att skydda processer i kernel-mode från varandra. En process i kernel mode har full tillgång till alla datastrukturer och adresser i hela systemet och kan enkelt gå runt de inbyggda säkerhetsmekanismerna.

5.2.1 Ett systemanrops väg genom systemet

Alla delar av operativsystemet exekverar inte i kernelmode stora delar är implementerade som DLLer som i sin tur anropar tjänster i kärnan om det behövs.

För att ta ett exempel så leder ett anrop av WIN32s funktion *ReadFile* så småningom till ett anrop av en rutin i kärnan för att läsa data från en fil. Denna rutin måste exekvera i kernelmode då den modifierar interna strukturer. En speciell instruktion för mode-byte utförs och systemet befinner sig i kernelmode. Ett byte av mode fångas av operativsystemet som ser att en systemtjänst anropas. Operativsystemet validierar sedan argumenten till den anropade funktionen och om allt är OK så exekveras sedan den anropade funktionen. Innan den anropande processen får tillbaka kontrollen byter man tillbaka till user-mode.

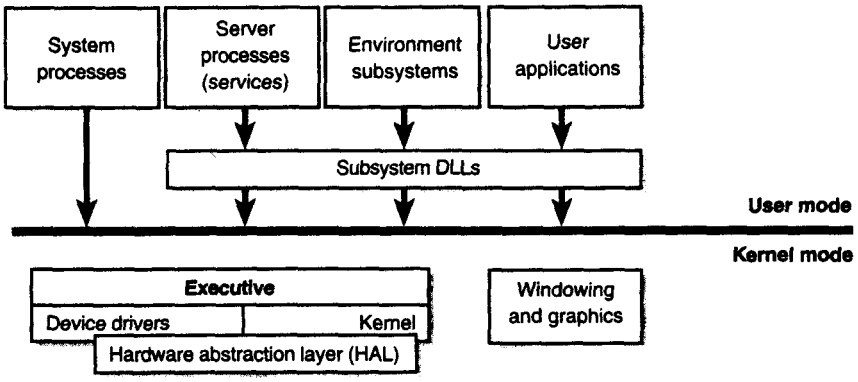
5.3 Arkitektur

I detta stycke presenteras Windows NT:s arkitektur.

5.3.1 Översiktligt

NT använder sig av en lagerindelad Client/Server arkitektur dvs man har delat upp systemet i två delar. En del exekverar i user-mode. Denna del erbjuder tjänster till andra program i systemet. Om tex tillgång till hårdvaran behövs så sker ett anrop till en annan del av kärnan som exekverar i kernelmode.

Varje del är i sin tur uppbyggd i lager så att varje lager bara behöver känna till gränssnittet för lagret under samt vilka tjänster den erbjuder till det närmaste lagret över.



Figur 6 - Översiktlig figur av Windows NTs arkitektur ***[Solomon 98]***

I figuren ser man ett det finns fyra typer av processer som exekverar i user-mode

Speciella stödprocesser för systemet som den process som har hand om inloggningar

Server processer, dvs systemtjänster som loggning av systemhändelser.

Systemmiljöprocesser, processer som tillhandahåller grundläggande operativsystemstjänster till användarens applikationer. Med NT följer 3 sådana miljöer Win32, POSIX och OS/2 1.2.

Användarens program som kan vara en av fem typer Win32, Windows 3.1, MS-DOS, POSIX, OS/2 1.2.

De olika processerna använder oftast inte de tjänster kärnan erbjuder direkt, utan detta sker via en eller flera system DLL-er. System DLL-ernas funktion är att översätta mellan ett officiellt och dokumenterat systemanrop till en eller flera anrop till kärnans ickeofficiella gränssnitt.

NTs kärna är i sin tur uppdelad på följande sätt. Executive ger de grundläggande tjänsterna man förväntar sig av ett operativsystem som minneshantering, process och trådhantering, säkerhet, I/O och IPC. Denna består i sin tur av två delar, kernel vilken ger operativsystemstjänsterna och device drivers vilka ger ett enhetligt gränssnitt mellan systemet och olika hårdvaruenheter som tex grafikkort, SCSI

kort. Dessa använder i sin tur ett abstraktionslager lagt på hårdvaran (hardware abstraction layer, HAL) för att isolera sig från plattformsspecifika detaljer.

I och med version 4 av NT flyttades också fönster och grafikhanteringen ned i kärnan och exekverar alltså i kernel-mode.

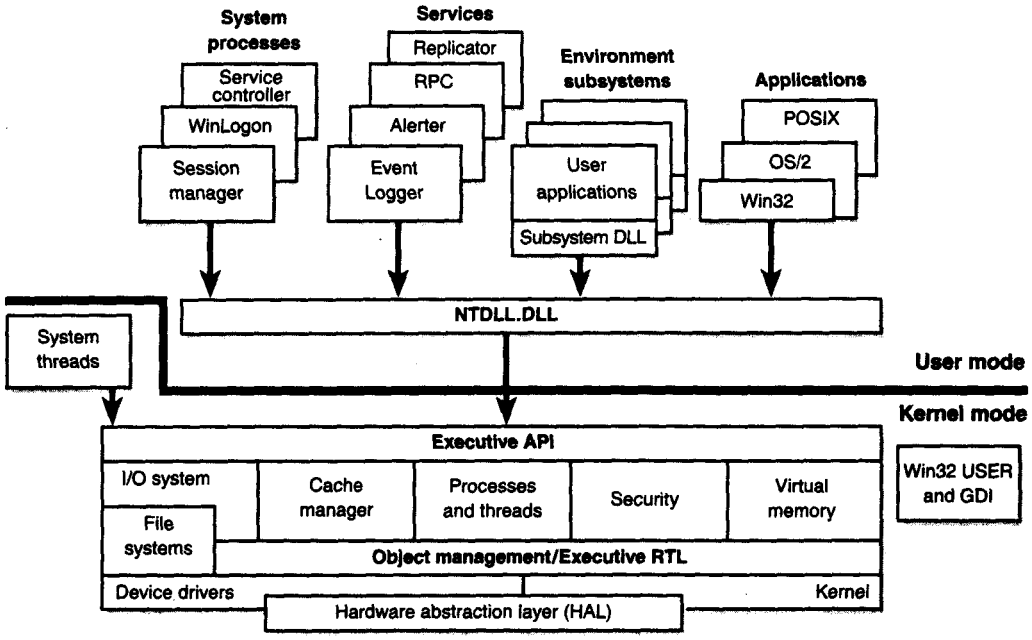
Genom denna lagerindelade design har man lyckats stödja ett flertal olika arkitekturer. Den första versionen stödde x86 och MIPS därefter följde snabbt även stöd för DEC Alpha AXP. I version 3.51 infördes stöd för Motorola PowerPC. Stödet för MIPS och PowerPC upphörde dock efter att version 4.0 släppts. NT 5 kommer endast att finnas för x86 och Alpha processorer. Eventuellt kommer också den nya familjen av processorer som utvecklas av Intel och Hewlett-Packard att stödjas (IA64 Intel Architecture 64).

Microsoft har officiellt sagt att NT kommer att förbättras och utökas för att på sikt ge ett riktigt 64-bitars programmeringsgränssnitt på både IA64 och Alpha.

För att stödja SMP på ett effektivt sätt så har kärnan designats för att låta flera processer samtidigt befinna sig i den dvs den är trådsäkert och har många synkroniseringspunkter i sig för att ge större effektivitet vid flertrådad exekvering.

5.3.2 In på djupet

En mer exakt bild av NTs inre struktur ges nedan.



Figur 7 - ***Windows NTs arkitektur [Solomon 98]***

5.3.2.1 Enviroment subsystems & subsystem DLLs

I NT finns tre olika enviroment subsystems, POSIX, Win32 & OS/2 (OS/2 finns endast på x86-plattformen). Av dessa har Win32 en särställning på det sättet att

NT inte kan köras utan den. Win32 körs alltså alltid medan de andra undersystemen bara startas om behov finns.

Ett subsystems roll är att erbjuda en applikation en uppsättning systemtjänster och en systemmiljö att exekvera i. De olika systemen ger tillgång till miljöer med olika uppförande och syntax för funktionsanropen. Varje applikation kan endast tillhöra ett undersystem, dvs en Win32 applikation kan inte använda FORK anropet i POSIX-miljön.

5.3.2.2 NTDLL.DLL

NTDLL är ett bibliotek som stödjer de olika subsystemen. För att undvika onödiga anrop till rutiner i kernel-space så har många run-time-library funktioner implementerats i denna DLL. Om ett anrop ner i kernel-mode är nödvändigt så exekverar denna DLL det anrop som leder till ett byte av mode och ett anrop till system-service-dispatchern.

5.3.2.3 Executive

Det övre lagret i kernelmode utgörs av Executive-komponenten. Den har bland annat till uppgift att vara ett gränssnitt mot user-mode. Den innehåller även funktioner som används av device-drivers dvs andra processer som exekverar i kernel-mode. Dessutom så innehåller den en uppsjö av odokumenterade funktioner som används internt mellan olika komponenter och internt i komponenterna.

5.3.2.4 Process & Thread manager

Skapar och terminerar processer och trådar. Den grundläggande funktionaliteten är implementerade i Kernel men denna komponent ger utökad semantik och fler funktioner.

5.3.2.5 Virtual memory manager

Denna komponent implementerar det virtuella minne som används. Den stödjer även cache-managern.

5.3.2.6 Security reference manager

Denna komponent ser till att den säkerhetspolicy som valts upprätthålls på den lokala datorn. Den vaktar operativsystemets resurser genom skydd och spårning (auditing).

5.3.2.7 I/O system

Ger enhets-oberoende in och ut operationer och ser till att rätt device driver anropas.

5.3.2.8 Cache manager

För att ge bra prestanda på diskoperationer så används en diskcache. Både skrivningar och läsningar cachas av denna komponent.

5.3.2.9 Kernel

Kärnan utför de mest grundläggande operationerna i systemet. Kärnan är den komponent som ligger längst ned i systemet närmast abstraktionslagret för hård-

varan och innehåller tex implementeringen av trådhantering, trap och undantagshantering, interupthantering, SMP-stöd.

Stora delar av kärnan kan inte swappas ut utan ligger alltid i internminnet och även om operationer i kärnan kan avbrytas av en ISR så kan en tråd i kärnan aldrig avbrytas av en annan tråd. Av denna anledning och för att få kärnan så liten som möjligt så sker inga kontroller av argumenten i kärnan utan man utgår från att den som gör anropet redan har verifierat parametrarna.

5.3.2.10 Hardware Abstraction Layer (HAL)

För att få ett system som är porterbart och isolera själva kärnan från skillnader mellan olika hårdvaruarkitekturer så har man lagt ett lager ovanpå själva hårdvaran. Detta lager ger ett gränssnitt på låg nivå till hårdvaran men man isolerar systemet från sådana saker som hantering av interupt, de mekanismer som används för SMP, IO-gränssnitt och så vidare. Det vill säga funktioner som inte bara är arkitekturspecifika utan också maskinspecifika. Valet av HAL görs när operativsystemet installeras och beror till exempel på vilken typ av moderkort som sitter i datorn.

5.3.2.11 Device drivers

Device drivers är laddbara moduler som exekverar i kernel-mode. De ger ett gränssnitt mellan IO-systemet och den hårdvara som ska användas. Men även dessa måste gå via HAL för anropen mot hårdvaran.

En device driver är enda sättet för en användare att få egen kod att exekvera i kernel-mode.

5.3.2.12 Session Manager (SMSS)

Sessions hanteraren är den första processen i user-mode som startas i systemet. Den initialiserar vissa portar och miljövariablerna. Laddar Win32 och startar subsystemprocesserna samt WINLOGON processen.

5.3.2.13 Logon (WINLOGON)

Logon hanterar inloggningar från interaktiva användare dvs användare som använder datorn direkt via tangentbordet. När användaren trycker Ctrl-Alt-Delete anropas WINLOGON. Användaren får sedan mata in användarnamn och lösenord, dessa kontrolleras sedan. Om lösenordet är korrekt så läses användarens inställningar upp ur registret och skalet EXPLORER startas.

5.3.2.14 Services

Services kan ses som NTs motsvarighet till UNIX demoner, dvs processer som startas automatiskt vid systemstart och som inte behöver någon speciell inloggning. De är egentligen helt vanliga Win32 program som använder vissa speciella anrop för att registrera sig hos service-controllern.

Många av NTs komponenter är implementerade som tjänster tex spoolern, eventloggen och vissa nätverkskomponenter.

5.4 Server vs workstation

NT finns i tre olika varianter *Windows NT Workstation*, *Windows NT Server* och *Windows NT Server, Enterprise Edition*. Vad skiljer dessa åt?

Till att börja med är de optimerade för olika uppgifter. Windows NT Server är optimerad för att vara en nätverksserver med hög prestanda medan Workstation är optimerad för att användas som en arbetsstation. De flesta filerna är likadana i de olika versionerna däremot har man ändrat parametrar i registret för schemaläggningen av trådar, diskcache-hantering odyl för att ge systemet de önskade egenskaperna.

Skillnaderna mellan workstation och server är också antalet samtidiga nätanslutningar till delade resurser som diskar och skrivare. En NT Workstation är begränsad till 10 samtidiga uppkopplingar till en delad resurs. Denna begränsning finns ej i NT Server. Det är även skillnad vad gäller SMP. En workstation stödjer två processorer, Server stödjer fyra och Server Enterprise Edition stödjer åtta processorer.

6 Viktiga mekanismer för ett realtidssystem

Detta kapitel försöker besvara frågan “Vilka mekanismer måste ett operativsystem erbjuda för att det ska vara möjligt att använda det för utveckling av realtidssystem?”.

Det vill säga om man studerar alla de tjänster operativsystemet erbjuder.Ar det några som är nödvändiga för att man på ett smidigt sätt ska kunna använda det vid utveckling av realtidssystem? Att det ska vara relativt enkelt att utveckla systemet är ett nödvändigt krav då man i teorin kan gå förbi hela operativsystemet och själv implementera de tjänster man behöver utan att använda något av operativsystemet. Men vi utgår från att man verkligen vill använda det valda operativsystemet.

Vi talar nu om mekanismer på en väldigt hög nivå. Det finns självklart en mängd ytterligare funktioner som är önskvärda eller som måste finnas sås om hantering av priority-inversion osv.

6.1 Multitasking och stöd för trådar

Multitasking och trådar är centrala i så gott som alla system som rör sig på en nivå över det mest triviala. Många problem blir mycket enklare att hantera vid trådning och systemets kapacitet används effektivare. Ett konkret exempel kan vara att man har en tråd som hanterar användargränsnittet och en annan som utför själva beräkningsarbetet. Användargränsnittet uppdateras då kontinuerligt och systemet uppfattas som mer responsivt.

6.2 Möjlighet att ha olika prioriteter på olika processer och trådar

Om man nu i systemet har flera trådar som har möjlighet att exekvera hur ska operativsystemet välja ut nästa tråd för körning? Vissa trådar kan tänkas vara viktigare än andra, om man tar ett exempel ur bilvärlden så är det antagligen viktigare att ABS-bromsarna fungerar än att klockan på instrumentpanelen visar exakt rätt tid.

Det krävs alltså en mekanism för att ge olika trådar olika prioriteter i systemet.

6.3 Prioritetsbaserad schemaläggning

Ska man ha någon nytta av att tilldela olika processer olika prioriteter så gäller det att dessa också används när det är dags att välja vilken process som ska köra härnäst. Det behovs alltså en algoritm för schemaläggning som tar hänsyn till processernas prioritet. En viktig egenskap är att en process på en högre prioritet idealt inte ska märka av att det finns processer på lägre prioritet. Processerna på lägre prioritet kan inte påverka dem på en högre nivå.

6.4 Synkronisering & kommunikation mellan processer

Vi har nu ett antal olika processer i systemet som kan avbryta varandra vid relativt godtyckliga tidpunkter enbart baserat på processernas prioriteter. Vad händer om en process blir avbruten precis när den har uppdaterat delar av en datastruktur och denna datstruktur används av den avbrytande processen? Om vi har flera processer i systemet som alla har olika roller så måste det finnas en metod för samordning och styrning alltså ett sätt för kommunikation mellan olika processer.

De mekanismer som är vanligast för att hantera detta är semaforer, meddelanden, händelser, critical-sections, mutex med mera, för en genomgång av dessa hänvisas till [Silberschatz & Galvin 94].

6.5 Dynamiskt minne

I ett system är ofta tillgången på minne begränsad. Det finns inte tillräckligt mycket för att statiskt allokera minne till alla processer. Det måste då finnas ett sätt att dynamiskt allokera minne som sedan lämnas tillbaka till systemet och som sedan kan användas av någon annan process.

6.6 Tidshantering

I många sammanhang är det viktigt att vissa operationer sker med en viss frekvens eller vid en speciell tidpunkt. Det kan till exempel röra sig om buffertar i hårdvara som måste tömmas med jämna mellanrum. Detta kräver att någon form av timers finns i systemet och att precisionen hos dessa är tillräckligt god för de syften de ska användas till.

7 Systemmekanismer i NT

I detta kapitel beskrivs några olika systemmekanismer i NT. Valet av mekanismer har gjorts med fokus på systemstabilitet och realtidsegenskaper. Detta kapitel bygger till stora delar på [Solomon 98].

7.1 Objekt

Internt i operativsystemet finns en rad olika datastrukturer som representerar till exempel semaforer, trådar och dylikt. I enklare operativsystem förekommer det att man låter systemanrop returnera en pekare till någon intern datastruktur och denna pekare sedan används vid olika systemanrop.

En risk med denna metod är att användaren eventuellt kan ändra värden i datastrukturerna och/eller skicka in felaktiga pekare, detta kan leda till ett system där stabilitet och säkerhet inte kan garanteras.

För att förhindra detta så införde man i NT en objektmodell. Då systemprogrammeringsspråket är C så är objektmodellen annorlunda än till exempel i C++. Objekten i systemet är inte äkta objekt med informationsgömning och olika metoder för att påverka den dolda informationen. NTs objektmodell bygger på att alla datastrukturer är gömda för användarens processer, all åtkomst sker via ett handtag, andra processer som exekverar i kernel-mode kan däremot direkt modifiera datastrukturerna. För att ytterligare skydda systemet från felaktiga user-mode applikationer verifieras parametrarna vid anrop till rutinerna i kernel-mode.

Den del av NT som hanterar objekten och alla tjänster runt dem är den så kallade *Object Manager*. Objektmodellen i NT designades för att uppfylla följande krav

- Ge en gemensam och enhetlig mekanism för användandet av systemresurser.
- Isolera skyddet av objekt till en del av operativsystemet så att säkerhetsklassningen C2 kan uppfyllas.
- Ge möjlighet att debitera olika trådar för dess användande av olika systemresurser.
- Ge ett system för namngivning i vilket man kan hantera olika enheter, filer, och andra typer av objekt på ett konsistent sätt.
- Ge stöd för de krav som kommer av möjligheten att stödja olika typer av applikationer såsom Win32, Win16, dos, posix OS/2.

7.2 Processer och trådar

I detta stycke redovisas hur NT hanterar processer och trådar. Särskild tyngd har lagts på schemaläggning och prioritetshantering.

7.2.1 Prioriteter

Tidigare slogs fast att prioritetsbegreppet är centralt. Hur hanterar då NT prioriteter?

7.2.1.1 Prioritetsnivåer från utvecklarens synpunkt

Ur en utvecklares synpunkt så har NT fyra olika basklasser av prioriteter som en process kan tillhöra. Dessa är i prioritetsordning

1. Realtid (real time)
2. Hög (high)
3. Normal
4. Låg (Idle)

Dessa prioritetsklasser är ett sätt att ange processers prioritet relativt varandra. I systemet finns ett antal systemprocesser som kör på hög basprioritet men det vanligaste är att processer har basprioritet Normal. Varje tråd i en process har sedan i sin tur en prioritet relativt andra trådar i processen. Dessa olika nivåer är i prioritetsordning

1. Time critical
2. Highest
3. Above normal
4. Normal
5. Below normal
6. Lowest
7. Idle

7.2.1.2 Prioritetsnivåer ur systemets synpunkt

I avsnitt 7.2.1.1: "Prioritetsnivåer från utvecklarens synpunkt" beskrevs utvecklarens vy på processer, trådar samt dess prioriteter. Internt i operativsystemet används en annan modell.

Internt har NT 32 olika prioritetsnivåer 0 till 31 där 0 är den lägsta prioriteten och 31 den högsta. Dessa prioritetsnivåer är uppdelade på följande sätt

- Realtidsnivåer (16 - 31)
- Variablanivåer (1 - 15)
- En systemnivå (0) som är reserverad för systemets *zero-page thread.*

En tråds interna prioritet i NT baseras på två olika faktorer: dels den prioritetsklass som dess process tillhör och dels den relativa prioritet inom processen som den har tilldelats. Dessa två faktorer vägs sedan samman och ger en intern prioritet enligt tabell 1.

	Real time	High	Normal	Idle
Time critical	31	15	15	15
Highest	26	15	10	6
Above normal	25	14	9	5
Normal	24	13	8	4
Below normal	23	12	7	3
Lowest	22	11	6	2
Idle	16	1	1	1

Tabell 1 - Mappning mellan basprocessens prioritetsklass (kolumn), trådens relativa prioritet (rad) och intern basprioritet.

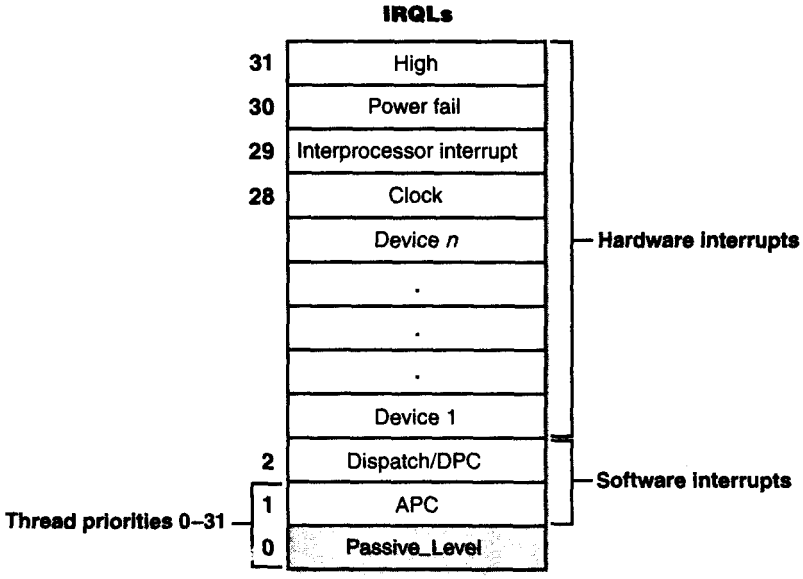
Alla trådar har två olika prioritetsvärden nuvarande- och bas-prioritet. Värdena i tabell 1 anger värdet på basprioriteten för olika kombinationer av basprocessens prioritetsklass och trådens relativa prioritet. Värdet på nuvarande-prioritet är det som används vid all schemaläggning av trådar.

Trådar som har en prioritet som tillhör de variabla nivåerna kan få sin nuvarande prioritet ändrad av schemaläggaren. Detta sker i en mängd olika fall, t ex när en tråd startas efter I/O så höjs dess nuvarande prioritet med allt från 1 till 8 steg. Den förhöjda prioriteten klingar sedan av med 1 enhet per timeslice tills tråden är tillbaka på sin basprioritet.

Trådar med prioriteter på någon av realtidsnivåerna behåller däremot alltid sin prioritet, då man antar att utvecklare som lagt sina trådar på så pass höga prioriteter har en anledning till det och vet vad de gör.

7.2.1.3 Interuptnivåer och prioritetsnivåer

Hur relaterar trådarnas prioritetsnivåer med prioritetsnivåerna i resten av systemet? Alla trådar i systemet exekverar antingen på IRQ-nivå 0 eller 1. Det normala är att trådarna exekverar på nivå 0 men vissa trådar i kärnan exekverar på nivå 1.

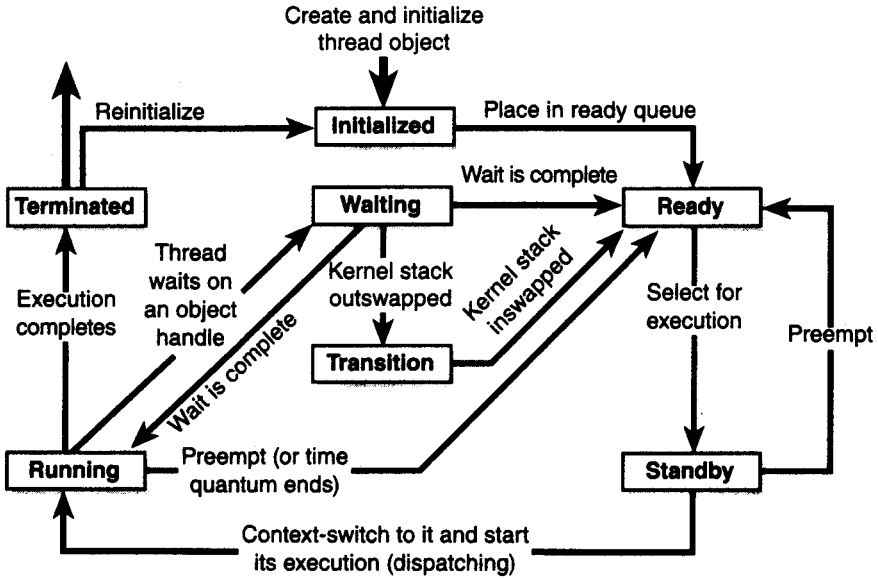


Figur 8 - Trådprioriteter relativt interuptnivåer ***[Solomon 98]***

Detta innebär att ingen tråd kan hindra hårdvaruinterupt från att behandlas. Schemaläggningen sköts av en process på nivå 2 vilken även kallas dispatch level.

7.2.1.4 En tråds tillstånd

För att förstå hur schemaläggningen av trådar sker behövs förståelse för de olika tillstånd som en process i NT kan befinna sig i.



Figur 9 - En tråds tillstånd ***[Solomon 98]***

En tråd befinner sig alltid i något av följande tillstånd:

- **Ready**: En tråd som är ready är färdig att exekvera, den väntar på att få tillgång till en CPU. När dispatchern letar efter en tråd att exekvera så väljs en av de trådar som är ready.
- **Standby**: En tråd som är standby har blivit utvald att exekvera på en viss processor, det enda den väntar på är en context-switch. Endast en tråd per processor kan befinna sig i standby-läge.
- **Running**: När väl dispatchern genomfört en context-switch till en tråd så går tråden till tillståndet running och exekverar. En tråd exekverar tills något av följande inträffar: kärnan avbryter den, en tråd med högre prioritet blir ready, dess tidskvanta tar slut, den avsäger sig processorn frivilligt, den terminerar.
- **Waiting**: Det finns flera anledningar för en tråd att vänta på något. Tråden kan frivilligt vänta på t ex en semafor eller något annat synkroniseringsobjekt, tråden kan vänta på operativsystemet (I/O), systemet kan även suspendera en tråd. När en tråd väntat klart så flyttas den beroende på prioritet till något av tillstånden running eller ready.
- **Transition**: Om en tråd är klar att exekvera men dess stack är utswappad så befinner sig tråden i tillståndet transition i väntan på att stacken ska swappas in i minnet. När stacken väl är tillbaka i minnet så hamnar tråden i tillståndet ready.
- **Terminated**: När en tråd exekverat klart så hamnar den i tillståndet terminated. Beroende på om andra system refererar till trådobjektet eller ej så förstörs i vissa fall objektet och i andra fall återanvänds det.

7.2.2 Schemaläggning

I detta stycke beskrivs hur schemaläggningen av processer i NT går till. För att kunna beskriva hur detta sker så behövs vissa begrepp definieras.

7.2.2.1 Exekveringskön

För varje prioritetsnivå finns det en kö med de trådar som är klara att exekvera dvs de befinner sig i tillståndet ready.

7.2.2.2 Tidskvanta

En tråds tidskvanta (quantum) är den tid som en tråd maximalt får exekvera innan NT ger en annan tråd med samma prioritet en chans att exekvera. Om en tråd har avslutat ett tidskvanta och det inte finns någon annan tråd med samma prioritet som är klar att exekvera så schemaläggs tråden för att exekvera ännu ett quantum.

Längden på ett tidskvanta varierar mellan olika trådar och olika maskiner. En mängd olika faktorer påverkar längden på ett tidskvanta:

- Varje tråd har ett heltalsvärde på antalet tidskvanta-enheter som tråden har tillgång till. Detta värde är ingen tid utan bara ett relativt mått på antalet enheter tråden får exekvera.

- Varje tråd får normalt till en början 6 tidskvanta enheter på NT Workstation och 32 enheter på NT Server. Tanken är att en tråd på NT server ska hinna exekvera klart under ett tidskvanta.
- Varje gång ett klockavbrott kommer dras 3 från trådens tidskvanta-värde. Om tråden hamnar på 0 eller färre enheter så avbryts tråden.
- Tiden mellan klockavbrotten varierar mellan de olika plattformarna. På vissa Intelsystem är det 10ms mellan avbrotten på andra system är det 15ms och på DEC Alpha AXP system är det 7.8125ms mellan avbrotten.
- På NT Workstation så kan en tråd temporärt få fler enheter om den tillhör den applikation som har fokus i användargränssnittet. Mellan 1 och 3 enheter kan adderas till grundvärdet.
- Antalet tidskvanta-enheter fördubblas när NT höjer en process prioritet för att undvika priorityinversion.
- När en tråd blir ready efter att ha väntat så får den ett antal extra enheter. Trådar med realtids prioritet (16 till 31) får sitt värde satt till standard värdet det vill säga tråden får en ny fräsch timeslice. Trådar med variabel prioritet (0 till 15) får sitt värde minskat med 1 och om resultatet blir 0 så får tråden sitt standardvärde av tidskvanta-enheter.

Processor	Klock intervall	Standard quantum på NT Workstation	Standard quantum på NT Server
486 baserat en-processors system	10 ms	20 ms	120 ms
Pentium och Pentium Pro baserat en-processors system	15 ms	30 ms	180 ms
486 baserade fler-processor system	10 ms	20 ms	120 ms
Övrig Intel baserade fler-processor system	15 ms	30 ms	180 ms
DEC AXP system	7.8ms	15.6 ms	93.6 ms

Tabell 2 - Arkitekturiella skillnader för standard-quantum längden.

7.2.2.3 Frivilligt överlämnande

Med frivilligt överlämnande (Eng. Voluntary Switch) menas att en tråd frivilligt avsäger sig processorn. Detta kan till exempel ske genom att tråden väntar på något synkroniseringsobjekt. En tråd som frivilligt avsäger sig processorn blir inte av med sina quantum-enheter utan den blir schablonmässigt av med en quantum-enhet för vänteoperationen och får dessutom den normala tilldelningen av quantum-enheter när den får processorn nästa gång.

7.2.2.4 Preemption

Preemption kan ses som raka motsatsen till frivilligt överlämnande. En tråd som råkar ut för detta blir av med processorn och hamnar i ready-tillståndet. Detta inträffar när en annan tråd med högre prioritet blir klar för att exekvera. Detta kan tex inträffa när en annan tråd med högre prioritet avslutar en väntan på något eller om prioriteteten för en tråd ändras. En tråd som avbryts hamnar inte sist i exekveringskön utan behåller sin plats först i kön och även de quantum-enheter den har kvar.

7.2.2.5 Quantum end

Enligt avsnitt 7.2.2.2: "Tidskvanta" så har varje tråd en övre gräns för den tid den får exekvera innan systemet ger en annan tråd en chans att exekvera, vad händer då när en tråds tidskvanta tagit slut? Först måste systemet avgöra om trådens prioritet ska förändras av någon anledning. Efter detta söker systemet efter den tråd i tillståndet ready och som har högst prioritet, Denna tråd väljs ut för att få exekvera.

7.2.2.6 Terminering

När en tråd terminerar antingen genom att den returnerar ett värde, genom ett anrop till *ExitThread* eller dödas med *TerminateThread*, flyttas den från tillståndet ready till tillståndet terminated. Om det inte finns några referenser till tråd-objektet så tas tråden bort och dess datastrukturer avallokeras.

7.2.3 Justering av schemaläggning

För att ge ett system som fungerar smidigt och som känns responsivt så justerar NT dynamiskt trådars prioriteter och storleken på dess quantum. NT rör aldrig prioriteterna för trådar med prioritetsnivå mellan 16 och 31 (real-tid).

Det finns fyra typer av schemaläggningsjusteringar, Quantum stretching, Priority boost efter I/O, Priority boost för trådar som väntar på Windowsmeddelanden och Priority boost för undvikande av utsvältning / priority-inversion.

7.2.3.1 Quantum stretching

I NT finns möjlighet att tilldela den applikation som har fokus i användargränssnittet (dvs är aktiv) större quantum än andra applikationer. Man har resonerat som så att den applikation som användaren just nu använder är lite viktigare än många andra processer i systemet.

Användaren kan själv bestämma hur detta ska ske genom en inställning i systemapplikationen i kontrollpanelen. Under fliken prestanda kan man välja hur mycket programmet i aktivt fönster ska accelereras. Tre olika inställningar finns

ingen, medel och maximal. Hur mycket extra tid som applikationen får ses i tabell 3.

Acceleration s inställning	System variabel	Windows NT Workstation	Windows NT Server
Ingen	PspForeGroundQuantum[0]	6	36
Medel	PspForeGroundQuantum[1]	12	36
Maximal	PspForeGroundQuantum[2]	18	36

Tabell 3 - Antal quantum-enheter för den applikation som har fokus

Som man kan se i tabellen så ignorerar NT Server denna inställning och ger alla applikationer lika mycket tid.

7.2.3.2 Priority boost efter I/O

När en tråd avslutat vissa I/O operationer så ger NT en temporär höjning av prioriteten, att vänta på en snabb enhet ger en liten ökning och att vänta på en långsam ger en stor ökning. Hur stor ökningen är finns för några enheter i tabell 4. Varje gång tråden sedan avslutar ett quantum så minskas dess prioritet med 1 tills den är nere på den ursprungliga prioritetsnivån.

Device eller objekt	Antal enheters ökning av prioriteten
Event, semafor	1
Disk, CD-ROM, parallellport, video	1
Nätverk, mailslot, named pipe, serieport	2
Tangentbord, mus	6
Ljud	8

Tabell 4 - Ökning av prioriteten efter väntan på enheter

Detta gäller enbart för trådar med prioriteter i det dynamisk området (0 till 15). Oavsett vilken prioritet en tråd i det dynamiska området har så kan dess prioritet inte höjas upp till realtidsområdet.

7.2.3.3 Priority boost för trådar som väntar på Windowsmeddelanden

En tråd som väntar på windowsmeddelanden eller indata från användaren får sin prioritet höjd till 14 samt ett quantum som är dubbelt så stort mot normalt. Detta får att tråden snabbt ska få behandla det meddelande det väntar på och dessutom ha extra lång tid på sig för själva behandlingen.

Till skillnad från prioritetshöjningarna vid I/O så sätts prioriteten till sitt normalvärde när tråden får ett nytt quantum.

7.2.3.4 Priority boost för undvikande av utsvältning / priority-inversion

Om man har flera trådar som oberoende av varandra allokerar och använder systemresurser så är det inte omöjligt att priority-inversion uppstår.

För att hantera detta har man valt att låta en systemtråd *balance set manager* gå igenom köerna med processer som är färdiga för exekvering och leta efter trådar som inte fått exekvera de senaste 300 klock-ticken (ca 3 till 4 sekunder). Dessa trådar får sin prioritet höjd till 15 och ett dubbelt så stort quantum som normalt. När de 2 quantumen är slut så återgår tråden omedelbart till sin ursprungliga prioritet.

I varje genomgång av köerna så kontrolleras bara 16 trådar per prioritetsnivå om det är fler trådar än så i kön så sparas positionen och genomgången fortsätter här nästa gång. När 10 trådar har fått ökad prioritet så slutar genomgången för att fortsätta på samma position nästa omgång.

7.2.4 Schemaläggning på fler-processor system

För att kunna beskriva hur NT hanterar schemaläggning på flerprocessors system så måste ett par termer definieras:

Affinitet: Varje tråd har en affinitetsmask som specifierar på vilka processorer som tråden tillåts exekvera. Det normala är att alla trådar kan exekvera på alla processorer, men masken kan ändras till exempel genom anrop till *SetProcessAffinityMask*.

Ideal och nästa processor: Varje tråd har två CPU-nummer: en ideal processor och en nästa processor. Den ideala processorn är den processor som tråden bör exekveras på. Nästa processor är den processor där tråden kommer att exekvera eller där den exekverades sist. Den ideala processorn väljs slumpmässigt när tråden skapas. NT ändrar inte värdet på en tråds ideala processor när väl tråden skapats, en tråd kan dock själv byta ideal processor genom systemanropet *SetThreadIdealProcessor*

När en tråd blir klar för exekvering (dvs hamnar i tillståndet ready) och flera processorer finns tillgängliga så försöker NT först schemalägga tråden på dess ideala processor. Om denna ej är ledig görs ett försök med trådens nästa processor och om den inte är ledig provas den processor som schemaläggaren exekverar på. Om ingen av dessa processorer är tillgängliga så används den lediga processor som har lägst CPU-nummer.

Om alla processorer är upptagna och en tråd blir klar för exekvering så försöker man avbryta en exekverande tråd på någon av processorerna. Först undersöks trådens ideala processor sedan trådens nästa processor. Om ingen av dessa finns med i trådens affinitetsmask så väljs den första processorn som tråden kan köra på.

Om den valda processorn redan kör en tråd eller har en tråd som är schemalagd och väntar på att få exekvera så jämförs prioriteterna. Om den nya tråden har en prioritet som är högre än den redan schemalagda så ersätts den gamla tråden av

den nya. Om prioriteten för den nya tråden är lägre än för den redan schemalagda så läggs den nya tråden i processorns exekveringskö.

Hittills har beskrivits hur en schemaläggning av en tråd som blir ready går till. Det finns även ett fall då en processor blir ledig genom att den exekverande tråden väntar på något, ändrar sin prioritet eller på något annat sätt avsäger sig processorn. I en-processor fallet så gick man helt enkelt igenom exekveringsköerna för att hitta den tråd som var klar att exekvera med högst prioritet så gör man inte i flerprocessorsfallet.

Den tråd som väljs för exekvering på en viss processor är den tråd som har högst prioritet och är klar att köra och som uppfyller ett av följande krav:

- Tråden exekverade senast på den aktuella processorn.
- Tråden har sin ideala processor satt till den aktuella processorn.
- Tråden har väntat på att få exekvera mer än 2 quantum.
- Tråden har en prioritet större än eller lika med 24.

Anledningen till att man prioriterar den processor där tråden sist exekverade är att chansen är stor att stora delar av tråden ligger kvar i processorns cache-minne, vilket ökar prestanda.

7.3 Avbrottshantering

Centralt i avbrottshanteringen är den så kallade *Interupt Service Routine* (ISR) vilket är den rutin som ska anropas då ett visst avbrott genererats. En ISR kan endast avbrytas av avbrott med högre prioritet, inkommande avbrott köas dock upp. För att det inte ska ta för lång tid innan avbrotten på lägre nivåer servas bör därför ISR-en vara så liten som möjligt.

Den ideala ISR-en är bara några få rader som flyttar lite data för att sedan signalera en semafor så att en annan icke ISR process kan fortsätta bearbetningen av informationen.

För att kunna erbjuda en portbar mekanism för avbrottshanteringen har man i NT skapat så kallade *interupt-objekt*. Dessa objekt finns i kärnan och innehåller all den information som kärnan behöver för att associera en ISR med det avbrott som den ska hantera. Fördelen med interupt-objekt är att man isolerar ISR-en från själva hanterandet av avbrottsvektorer och andra arkitekturspecifika detaljer. En annan fördel är att flera olika objekt kan vara anslutna till ett och samma avbrott, man har alltså möjlighet att dela avbrott mellan olika enheter.

Microsoft rekomenderar att den längsta tid en ISR får ta är 25 µs. Detta är tyvärr något de inte själva följer särskilt väl. Mätningar på den drivrutin som hanterar hårddiskar anslutna via ATAPI befinner sig i bland upp till ca 5ms i sin ISR vilket är cirka 200 gånger mer än rekomenderat.

7.4 Multimedia timers

Hur multimedia timers är implementerade beskrivs bland annat i artikeln [Jones & Regehr 98] vilken sammanfattas i detta stycke.

7.4.1 Hårdvaran bakom timeravbrott

För att förstå multimediatimers är vi tvungna att studera hur hårdvaruplattformen fungerar. I NT ligger hårdvaran abstraherad under en HAL (Hardware Abstraction Layer). Olika arkitekturer har olika HAL. Vissa datortillverkare skickar med egna HAL specialgjorda för deras system.

Normalt ger ett HAL tillgång till periodiska avbrott. På HALX86 ligger periodtiden normalt mellan 10 ms och 15 ms. Genom ett anrop har kärnan möjlighet att ställa om periodtiden, detta är HAL-beroende men normalt kan tiden ställas mellan 1ms och 15 ms.

Ett konkret exempel på skillnaden mellan olika HAL är att HALX86 använder klockchipet 8254 på interupt 1 för timergenerering men HALMPS använder RTC på interupt 8. Olika HAL har också olika gränser som periodtiden kan variera mellan. På vissa HAL kan den inte ändras överhuvudtaget utan är satt till ett konstant värde.

7.4.2 Implementationen

Multimediatimers implementeras normalt med en tråd på hög prioritet som sätter en timer i kärnan för att sedan blockera på denna, när tråden väcks exekveras eventuellt en callback-funktion timern sätts på nytt och processen somnar sedan.

Följande algoritm beskriver mer detaljerat vad timerprocessen gör varje gång den anropas. En timer vet vilken periodtid den har, när den ska lösa ut nästa gång och vilken callback funktion som ska anropas. Timern använder följande algoritm.

1. Vänta på en anrop från kärnans timer.
2. Är den aktuella tiden större än den tidpunkt vi skulle löst ut? Om inte gå till 1.
3. Öka på tidpunkten för alarm med periodtiden, detta ger nästa alarmtidpunkt.
4. Anropa callbackfunktionen. Har tidpunkten för nästa alarm varit? Om ja gå till 3.
5. Begär en ny timer från kärnan, denna timer ska lösa ut på aktuell tidpunkt + 1000 (periodtiden).
6. Gå till 1.

Den timer i kärnan som används är implementerad på följande sätt

1. Avrunda tidpunkten för alarm till närmaste hela millisekund
2. Vänta på ett timerinterupt.
3. Har tidpunkten varit? Om ej gå till 2
4. Lös ut timern.

Om den periodtid man vill ha på sin multimediatimer passar dåligt ihop med interuptets periodtid från HAL kan konstiga effekter uppstå.

Om man till exempel har ett multiprocessor system som använder HALMPS så kan man bara sätta interuptfrekvensen till potenser av 2 dvs det närmaste 1000 Hz vi kommer är 1024 Hz vilket ger en periodtid på 976 us.

Antag nu att vi vill ha en multimediatimer med en periodtid på 1 ms. Detta leder till följande beteende.

Tidpunkt [us]	Timer interupt	Kerneltimer	Multimedia timer
0	Går	Sätts till 1000	Sätts till 1000
976	Går	t < 1000 inget görs	Väntar
1952	Går	t > 1000, Multimedia timer anropas ny timer 3000 (1952 + 1000 avrundat)	1000 har varit callback anropas. Ny timer 2000
2928	Går	t < 3000	Väntar
3904	Går	t > 3000 Multimedia timer anropas ny timer 5000 (3904 + 1000 avrundat)	2000 har varit, call-back anropas. Ny timer 3000. 3000 har varit callback anropas. ny timer 4000.
4880	Går	t < 5000	Väntar
5856	Går	t > 5000 Multimedia timer anropas ny timer 7000 (5856 + 1000 avrundat)	4000 har varit, call-back anropas. Ny timer 5000. 5000 har varit callback anropas. Ny timer 6000.

Tabell 5 - Exempel på oönskat beteende hos multimediatimers

I och med avrundningen till hela millisekunder och att timerinteruptets periodtid är mindre än multimediatimern så får vi ett fall där multimediatimern missar ett alarm och sedan kompenserar för detta genom att lösa ut två gånger vid nästa tillfälle. Skulle man studera medelvärdet på tiden mellan alarmen skulle det se bra ut snittet skulle ligga nära 1 ms om man däremot t ex skulle studera standardavvikelsen skulle man se att den var större än förväntat då alla periodtider avviker mycket från medelvärdet.

8 Testbeskrivning

Testerna i detta kapitel är inriktade på de delar som identifierades som extra viktiga i kapitel 6: "Viktiga mekanismer för ett realtidssystem".

I detta kapitel refereras även till vissa speciella funktioner i programmeringsgränssnittet Win32. För information om detta hänvisas till någon av de otaliga böcker som beskriver Win32 programmering.

8.1 Testernas utförande

Detta stycke beskriver den metod som använts för att mäta prestanda på vissa utvalda mekanismer.

8.1.1 Allmän beskrivning

Samtliga tester följer i stora drag samma struktur nämligen att den aktuella mätningen upprepas på olika prioritetsnivåer samt såväl på belastat som obelastat system.

Grundtanken bakom detta är att få fram mätvärden på variationen i svarstid under olika förhållanden. Den absoluta tiden är starkt beroende av det system man mäter på. Variationen kan däremot kopplas till de förändringar i omgivningen som sker under testen.

8.1.2 Mätmetod

För att utföra mätningar används systemets *PerformanceCounter*. Detta är en 64-bitars räknare i hårdvaran som kontinuerligt räknas upp. Frekvensen på uppräkningarna är plattformsberoende. På en uni-processors x86 maskin är frekvensen 1193182 Hz vilket ger en upplösning på ca 883ns, på en multi-processors x86 maskin är frekvensen samma som processorernas klockfrekvens. Man får alltså en högre precision på ett multi-processor system. Man ska dock vara medveten om att det krävs en I/O instruktion för att hämta värdet hos räknaren.Den verkliga precisionen är därför lägre än upplösningen på klockan.

Det går dessutom åt två systemanrop för att mäta hur lång tid en operation tagit. Dessa anrop påverkar självklart resultatet. I vissa fall kan det dessutom förekomma att testprogrammet blir avbrutet av någon systemprocess mitt i mätslingan vilket ger felaktiga mätvärden.

För att undvika att resultaten påverkas allt för mycket av omständigheter utom kontroll så upprepas varje mätning 10 000 gånger. Fördröjningar är dessutom införda i testprogrammen så att ett fullständigt test alltid tar ca 3 timmar (detta gäller självklart inte timertesterna).

Då det är variationen i tidsåtgång vid de olika testfallen som är intressant så bedöms denna mätmetod tillräckligt god för att ge användbara mätresultat trots de olika faktorer som sänker mätningarnas precision.

8.1.3 Belastningssimulering

För att belasta systemet under testerna användes benchmarkprogrammet WinBench 97 från ZDNet. Benchmarkprogrammet kan hämtas hem från ZDNets benchmarking hemsida på *http://www.zdnet.com/zdbop/*. Detta program genomför en rad prestandatester på tex processor, minne, hårddisk och grafik. Under prestandatesterna belastas systemet maximalt.

Då benchmarkprogrammet följer en viss cykel under testerna (först cpu, sedan disk, sedan video osv) så är det viktigt att benchmarkprogrammet hinner igenom lika många faser vid varje test. Detta för att förhindra att t ex ett snabbt program enbart utsätts för cpu-belastning medan ett något långsammare program både får cpu och disk belastning.

För att undvika detta så har fördröjningar lagts till i mätslingan utanför den del vars prestanda ska mätas. Tidsfördröjningen har valts till 0.36 s. Detta medför att en testomgång med 10000 värden tar minst en timme att utföra. En cykel av benchmarkprogrammet tar ca 45 minuter. På denna tid har alltså benchmark programmet hunnit köra igenom hela sitt test minst en gång.

Användandet av ett kommersiellt benchmark-program medför att samma belastning är möjlig att uppnå i flera olika testfall samt att belastningen är omväxlande och omfattande.

Nackdelen med att använda WinBench är att det i det belastade fallet inte är möjligt att göra en koppling mellan testresultat och belastning. Man vet inte om det är testerna av hårddisken eller grafikkortet som gett upphov till de fenomen man ser i resultaten.

8.1.4 Påverkande faktorer

När man normalt utvecklar realtidssystem har man ofta kontroll över de systemprocesser som finns i systemet. Systemprocessernas uppträdande är ofta väldokumenterat med avseende på belastning, prioritetsnivåer, antal.

För NT finns ingen lättillgänglig dokumentation över detta. Vilka processer som finns varierar dessutom mellan olika system beroende på vilka systemtjänster som installerats.

Detta innebär att förutom testprogrammet och benchmarkprogrammet finns en mängd andra processer med varierande prioriteter och dessa processer inverkar på testresultaten. Denna inverkan är i stort sett lika under samtliga tester så påverkan på testresultaten är antagligen liten.

8.2 Beskrivning av test för timerfunktioner

I NT finns två olika timermekanismer, dels vanliga och dels sk multimedia timers. Multimedia timers använder en separat process och lastar ned systemet rejält om en timer med kort periodtid används.

Samtliga tester utföres med programmet i tre olika basprioriteter normal, hög och realtid. Testerna görs obelastade, samt med systemet nedlastat med WinBench.

I Win32 finns en timermekanism som efter en viss tid skickar en händelse WM_TIMER till den applikation som skapat timern. Meddelandet behandlas sedan i en händelsehanterare i applikationen.

Det finns även möjlighet att få en funktion anropad när tiden löpt ut, men den mekanismen bygger på att operativsystemet tar hand om WM_TIMER-meddelandet för att sedan anropa den angivna funktionen. Man får då ännu ett lager mellan applikationen och timern vilket leder till sämre precision. Testerna utförs därför med meddelandebaserade timers.

Multimedia Timers är den timertyp som har den bästa noggrannheten i NT. Microsoft säger själva i Microsoft Developers Network (*http://msdn.microsoft.com*)

"Multimedia timer services allow applications to schedule timer events with the greatest resolution (or accuracy) possible for the hardware platform. These multimedia timer services allow you to schedule timer events at a higher resolution than other timer services."

Multimedia timers anropar en callback-funktion när den löser ut.

8.2.1 Normal timer

Mäta variationen i tidsfördröjningar implementerade med vanliga timers.

Beskrivning:

1. Detta test upprepas med frekvenser på 1, 10, 100 Hz.
2. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
3. En timer skapas med angiven periodtid.
4. I meddelandehanteraren för WM_TIMER meddelanden sparas värdet av high performance räknaren. Mätvärdena sparas i minnet för att i så liten utsträckning som möjligt påverka systemet.
5. När 10 000 värden samlats in så stoppas timern, värdena sparas till disk och programmet terminerar

Resultat:

Samplingarna subtraheras parvis så att antalet tick mellan två anrop till timerfunktionen fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.2.2 Multimedia Timers

Mäta variationen i tidsfördröjningar implementerade med multimedia timers.

Beskrivning:

1. Detta test upprepas med frekvenser på 1, 10, 100 Hz.
2. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
3. En multimedia timer skapas och dess upplösning sätts till 0 för att få periodtider med största möjliga precision.

4. I den funktion som timern anropar varje gång den utlöses sparas värdet av high performance räknaren. Mätvärdena sparas i minnet för att i så liten utsträckning som möjligt påverka systemet.
5. När 10 000 värden samlats in så stoppas timern, värdena sparas till disk och programmet terminerar

Resultat:

Samplingarna subtraheras parvis så att antalet tick mellan två anrop till timerfunktionen fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.3 Beskrivning av tester för meddelandehantering

Meddelanden är en av de centrala mekanismerna i IPC då man har möjlighet att dels väcka en tråd då något inträffat och dessutom att överföra information i meddelandet. I detta fall har den tråd som väntar på meddelanden en lägre prioritet än sändaren.

8.3.1 Tid för att skicka ett meddelande utan preemption

Mäta hur lång tid det tar att skicka ett meddelande samt hur lång tid det tar för ett meddelande att komma fram. Samt hur mycket denna tid kan fås att variera vid olika yttre omständigheter.

Beskrivning:

Huvudprogram:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. En tråd skapas i suspenderat läge med *CreateThread.*
3. Den nyskapade trådens prioritet sätts till THREAD_PRIORITY_LOWEST.
4. Ett event-objekt skapas med *CreateEvent(NULL, false, false, "SyncEvent").*
5. Tråden väcks med *ResumeThread*
6. Vänta på *SyncEvent* via *WaitForSingleObject.*
7. Värdet på high performance räknaren hämtas.
8. *PostThreadMessage* används för att skickar ett meddelande *BR_MSG_TEST_MESSAGE* till tråden.
9. Värdet på high performance räknaren hämtas.
10. Tidsåtgångarna beräknas och lagras i minnet.
11. Om 10 000 värden inte samlats in hoppa till 6.
12. Meddelandet *BR_MSG_QUIT_MESSAGE* skickas till tråden.
13. Mätvärdena sparas på disk.
14. Programmet terminerar.

Den skapade tråden:

1. Skapa en meddelandekö genom ett anrop till *PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE).*
2. Signalera *SyncEvent.*
3. Vänta på ett nytt meddelande mha *WaitMessage.*
4. Hämta meddelandet mha *GetMessage.*
5. Värdet på high performance räknaren hämtas.
6. Om meddelandet är WM_QUIT, avsluta tråden.
7. Signalera SyncEvent.
8. Hoppa till 3.

Resultat:

Resultatet är mätvärden på hur många tick det tog från att meddelandet skickades tills att huvudtråden exekverade igen. Tiden för meddelandet att nå fram till den nyskapade tråden mäts också.

På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.3.2 Tid för att skicka ett meddelande med normal schemaläggning

Mäta hur lång tid det tar att skicka ett meddelande samt hur lång tid det tar för ett meddelande att komma fram. Samt hur mycket denna tid kan fås att variera vid olika yttre omständigheter.

Beskrivning:

Huvudprogram:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. En tråd skapas i suspenderat läge med *CreateThread*
3. Den nyskapade trådens prioritet sätts till THREAD_PRIORITY_NORMAL.
4. Ett event-objekt skapas med *CreateEvent(NULL, false, false, "SyncEvent").*
5. Tråden väcks med *ResumeThread.*
6. Vänta på *SyncEvent* via *WaitForSingleObject.*
7. Värdet på high performance räknaren hämtas.
8. *PostThreadMessage* används för att skickar ett meddelande *BR_MSG_TEST_MESSAGE* till tråden.
9. Värdet på high performance räknaren hämtas.
10. Tidsåtgångarna beräknas och lagras i minnet.
11. Om 10 000 värden inte samlats in hoppa till 6.
12. Meddelandet *BR_MSG_QUIT_MESSAGE* skickas till tråden.
13. Mätvärdena sparas på disk.

14. Programmet terminerar.

Den skapade tråden:

1. Skapa en meddelandekö genom ett anrop till *PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE).*
2. Signalera *SyncEvent.*
3. Vänta på ett nytt meddelande mha *WaitMessage.*
4. Hämta meddelandet mha *GetMessage.*
5. Värdet på high performance räknaren hämtas.
6. Om meddelandet är WM_QUIT, avsluta tråden.
7. Signalera SyncEvent.
8. Hoppa till 3.

Resultat:

Resultatet är mätvärden på hur många tick det tog från att meddelandet skickades tills att huvudtråden exekverade igen. Tiden för meddelandet att nå fram till den nyskapade tråden mäts också upp.

På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.3.3 Tid för att skicka ett meddelande med preemption

Mäta hur lång tid det tar att skicka ett meddelande samt hur lång tid det tar för ett meddelande att komma fram. Samt hur mycket denna tid kan fås att variera vid olika yttre omständigheter.

Beskrivning:

Huvudprogram:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. En tråd skapas i suspenderat läge med *CreateThread.*
3. Den nyskapade trådens prioritet sätts till THREAD_PRIORITY_HIGHEST.
4. Ett event-objekt skapas med *CreateEvent(NULL, false, false, "SyncEvent").*
5. Tråden väcks med *ResumeThread.*
6. Vänta på *SyncEvent* via *WaitForSingleObject.*
7. Värdet på high performance räknaren hämtas.
8. *PostThreadMessage* används för att skickar ett meddelande *BR_MSG_TEST_MESSAGE* till tråden.
9. Värdet på high performance räknaren hämtas.
10. Tidsåtgångarna beräknas och lagras i minnet.
11. Om 10 000 värden inte samlats in hoppa till 6.

12. Meddelandet *BR_MSG_QUIT_MESSAGE* skickas till tråden.
13. Mätvärdena sparas på disk.
14. Programmet terminerar.

Den skapade tråden:

1. Skapa en meddelandekö genom ett anrop till *PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE).*
2. Signalera *SyncEvent.*
3. Vänta på ett nytt meddelande mha *WaitMessage.*
4. Hämta meddelandet mha *GetMessage.*
5. Värdet på high performance räknaren hämtas.
6. Om meddelandet är WM_QUIT, avsluta tråden.
7. Signalera SyncEvent.
8. Hoppa till 3.

Resultat:

Resultatet är mätvärden på hur många tick det tog från att meddelandet skickades tills att huvudtråden exekverade igen. Tiden för meddelandet att nå fram till den nyskapade tråden mäts också upp.

På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.3.4 Tid för att kontrollera om något meddelande finns i kö

Hur lång tid tar det att asynkront kontrollera om något meddelande finns i kön?

Beskrivning:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Skapa en meddelandekö genom ett anrop till *PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE).*
3. Värdet på high performance räknaren hämtas.
4. PeekMessage görs för att se om något meddelande finns i kön.
5. Värdet på high performance räknaren hämtas.
6. Tidsåtgången beräknas och lagras i minnet.
7. Om 10 000 värden samlats in sparas mätvärdena till disk och programmet terminerar.
8. En *Sleep* på 0.1 sekunder görs.
9. Gå till 3.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att titta i kön fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.4 Eventhantering

Skillnaden mellan meddelanden och events är att ett event inte innehåller någon information.

8.4.1 Tid för att sätta en event som ingen väntar på

Hur lång tid tar det att sätta en event som ingen väntar på dvs NT behöver inte göra någon schemaläggning eller dylikt.

Beskrivning:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Skapa ett event-objekt med *CreateEvent(NULL, true, false, "TestEvent").*
3. *ResetEvent* görs på TestEvent.
4. Värdet på high performance räknaren hämtas.
5. *SetEvent* görs på TestEvent.
6. Värdet på high performance räknaren hämtas.
7. Tidsåtgången beräknas och lagras i minnet.
8. Om 10 000 värden samlats in så sparas mätvärdena till disk och programmet terminerar.
9. En *Sleep* på 0.1 sekunder görs.
10. Gå till 3.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att sätta ett event fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.4.2 Tid för att sätta en event som någon tråd väntar på, utan preemption

Hur lång tid tar det att sätta en event om en process blir ready?

Beskrivning:

Huvudprogrammet:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Ett event-objekt skapas via *CreateEvent(false, false, "SyncEvent").*
3. Ett event-objekt skapas via *CreateEvent(false, false, "TestEvent").*
4. En tråd skapas i suspenderat läge med *CreateThread.*
5. Den nyskapade trådens prioritet sätts till THREAD_PRIORITY_LOWEST .
6. Tråden väcks med *ResumeThread.*
7. Vänta på *SyncEvent.*

8. Värdet på high performance räknaren hämtas.
9. *SetEvent* görs på TestEvent.
10. Värdet på high performance räknaren hämtas.
11. Tidsåtgången beräknas och lagras i minnet.
12. Om 10 000 värden samlats in dödas alla trådar, mätvärdena sparas till disk och programmet termineras.
13. Hoppa till 7.

Den skapade tråden:

1. Signalera *SyncEvent.*
2. Vänta på TestEvent via *WaitForSingleObject*.
3. Värdet på high performance räknaren hämtas.
4. *SetEvent* görs på SyncEvent.
5. Hoppa till 2.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att sätta ett event fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.4.3 Tid för att sätta en event som någon tråd väntar på med normal schemaläggning

Hur lång tid tar det att sätta en event om en process som har samma prioritet som den process som skickade eventet blir ready?

Beskrivning:

Huvudprogrammet:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Ett event-objekt skapas via *CreateEvent(false, false, "SyncEvent").*
3. Ett event-objekt skapas via *CreateEvent(false, false, "TestEvent").*
4. En tråd skapas i suspenderat läge med *CreateThread.*
5. Den nyskapade trådens prioritet sätts till THREAD_PRIORITY_NORMAL.
6. Tråden väcks med *ResumeThread.*
7. Vänta på *SyncEvent.*
8. Värdet på high performance räknaren hämtas.
9. *SetEvent* görs på TestEvent.
10. Värdet på high performance räknaren hämtas.
11. Tidsåtgången beräknas och lagras i minnet.

12. Om 10 000 värden samlats in dödas alla trådar, mätvärdena sparas till disk och programmet termineras.
13. Hoppa till 7.

Den skapade tråden:

1. Signalera *SyncEvent.*
2. Vänta på TestEvent via *WaitForSingleObject*.
3. Värdet på high performance räknaren hämtas.
4. *SetEvent* görs på SyncEvent.
5. Hoppa till 2.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att sätta ett event fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.4.4 Tid för att sätta en event som någon tråd väntar på, med preemption

Hur lång tid tar det att sätta en event om en process som har högre prioritet än den process som skickade eventet blir ready?

Beskrivning:

Huvudprogrammet:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Ett event-objekt skapas via *CreateEvent(false, false, "SyncEvent").*
3. Ett event-objekt skapas via *CreateEvent(false, false, "TestEvent").*
4. En tråd skapas i suspenderat läge med *CreateThread.*
5. Den nyskapade trådens prioritet sätts till THREAD_PRIORITY_HIGHEST.
6. Tråden väcks med *ResumeThread.*
7. Vänta på *SyncEvent.*
8. Värdet på high performance räknaren hämtas.
9. *SetEvent* görs på TestEvent.
10. Värdet på high performance räknaren hämtas.
11. Tidsåtgången beräknas och lagras i minnet.
12. Om 10 000 värden samlats in dödas alla trådar, mätvärdena sparas till disk och programmet termineras.
13. Hoppa till 7.

Den skapade tråden:

1. Signalera *SyncEvent.*

2. Vänta på TestEvent via *WaitForSingleObject*.
3. Värdet på high performance räknaren hämtas.
4. *SetEvent* görs på SyncEvent.
5. Hoppa till 2.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att sätta ett event fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.5 Signalerande och väntande på semaforer

Semaforer används för att synkronisera flera processer vid användandet av delade resurser som tex gemensamt minne eller någon hårdvaruenhet.

8.5.1 Tid för att få besked om att en semafor är signalerad

Hur lång tid tar det att få reda på att en semafor är signalerad om man inte väntar på den?

Beskrivning:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Skapa ett semafor-objekt med *CreateSemaphore(NULL, 0, 1, "TestSemafor").*
3. Värdet på high performance räknaren hämtas.
4. *WaitForSingleObject* med *dwMilliseconds* satt till 0 görs på semaforen.
5. Värdet på high performance räknaren hämtas.
6. Tidsåtgången beräknas och lagras i minnet.
7. Om 10 000 värden samlats in så sparas mätvärdena till disk och programmet terminerar.
8. En *Sleep* på 0.1 sekunder görs.
9. Gå till 3.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att hämta semaforens tillstånd fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.5.2 Tid för att släppa en semafor, en väntande tråd blir ready ingen preemption

Hur lång tid tar det att signalera en semafor om en annan tråd väntar på den och den väntande tråden har lägre prioritet än den signalerande tråden?

Beskrivning:

Huvudprogrammet:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Ett event-objekt skapas via *CreateEvent(false, false, "SyncEvent").*
3. Skapa ett semafor-objekt med *CreateSemaphore(NULL, 0, 1, "TestSemafor").*
4. En tråd skapas i suspenderat läge med *CreateThread.*
5. Den nyskapade trådens prioritet sätts till THREAD_PRIORITY_LOWEST.
6. Tråden väcks med *ResumeThread.*
7. Vänta på *SyncEvent.*
8. Värdet på high performance räknaren hämtas.
9. *ReleaseSemaphore* görs på *TestSemafor.*
10. Värdet på high performance räknaren hämtas.
11. Tidsåtgångarna beräknas och lagras i minnet.
12. Om 10 000 har samlats in så dödas tråden och mätvärdena sparas till disk..
13. Gå till 7.

Den skapade tråden:

1. Signalera på *SyncEvent.*
2. Vänta på *TestSemafor* via *WaitForSingleObject.*
3. Spara värdet på high performance räknaren.
4. Sätt *TestSemafor* till 0.
5. Signalera på *SyncEvent.*
6. Gå till 2.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att signalera semaforen samt starta tråden fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.5.3 Tid för att släppa en semafor, en väntande process blir ready ingen preemption

Hur lång tid tar det att signalera en semafor om en annan tråd väntar på den och den väntande tråden har samma prioritet som den signalerande tråden?

Beskrivning:

Huvudprogrammet:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Ett event-objekt skapas via *CreateEvent(false, false, "SyncEvent").*

3. Skapa ett semafor-objekt med *CreateSemaphore(NULL, 0, 1, "TestSemafor").*
4. En tråd skapas i suspenderat läge med *CreateThread.*
5. Den nyskapade trådens prioritet sätts till THREAD_PRIORITY_NORMAL.
6. Tråden väcks med *ResumeThread.*
7. Vänta på *SyncEvent.*
8. Värdet på high performance räknaren hämtas.
9. *ReleaseSemaphore* görs på *TestSemafor.*
10. Värdet på high performance räknaren hämtas.
11. Tidsåtgångarna beräknas och lagras i minnet.
12. Om 10 000 har samlats in så dödas tråden och mätvärdena sparas till disk.
13. Gå till 7.

Den skapade tråden:

1. Signalera på *SyncEvent.*
2. Vänta på *TestSemafor* via *WaitForSingleObject.*
3. Spara värdet på high performance räknaren.
4. Sätt *TestSemafor* till 0.
5. Signalera på *SyncEvent.*
6. Gå till 2.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att signalera semaforen samt starta tråden fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.5.4 Tid för att släppa en semafor, en väntande process blir ready med preemption

Hur lång tid tar det att signalera en semafor om en annan tråd väntar på den och den väntande tråden har högre prioritet än den signalerande tråden?

Beskrivning:

Huvudprogrammet:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Ett event-objekt skapas via *CreateEvent(false, false, "SyncEvent").*
3. Skapa ett semafor-objekt med *CreateSemaphore(NULL, 0, 1, "TestSemafor").*
4. En tråd skapas i suspenderat läge med *CreateThread.*
5. Den nyskapade trådens prioritet sätts till THREAD_PRIORITY_HIGHEST.

6. Tråden väcks med *ResumeThread.*
7. Vänta på *SyncEvent.*
8. Värdet på high performance räknaren hämtas.
9. *ReleaseSemaphore* görs på *TestSemafor.*
10. Värdet på high performance räknaren hämtas.
11. Tidsåtgångarna beräknas och lagras i minnet.
12. Om 10 000 har samlats in så dödas tråden och mätvärdena sparas till disk.
13. Gå till 7.

Den skapade tråden:

1. Signalera på *SyncEvent.*
2. Vänta på *TestSemafor* via *WaitForSingleObject.*
3. Spara värdet på high performance räknaren.
4. Sätt *TestSemafor* till 0.
5. Signalera på *SyncEvent.*
6. Gå till 2.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att signalera semaforen samt starta tråden fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.6 Minnesallokering

I NT finns två typer av minne man kan allokera dynamiskt dels det vanliga minnet vilket endast är åtkomligt i den processrymd man befinner sig och dels delat minne viket är åtkomligt för flera olika processer.

Delat minne används ofta för kommunikation med drivrutiner och mellan processer.

8.6.1 Tid för allokering och deallokering av minne

Hur lång tid tar det att allokera och deallokera minne? Tar det olika mycket tid för olika storlekar på det allokerade minnet?

Beskrivning:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Värdet på high performance räknaren hämtas.
3. *alloc* används för att allokera 2k minne.
4. Värdet på high performance räknaren hämtas.
5. Tidsåtgången beräknas och lagras i minnet.
6. Värdet på high performance räknaren hämtas.

7. *alloc* används för att allokera 4k minne.
8. Värdet på high performance räknaren hämtas.
9. Tidsåtgången beräknas och lagras i minnet.
10. Värdet på high performance räknaren hämtas.
11. *alloc* används för att allokera 8k minne.
12. Värdet på high performance räknaren hämtas.
13. Tidsåtgången beräknas och lagras i minnet.
14. Värdet på high performance räknaren hämtas.
15. *free* används för att deallokera 2k bufferten.
16. Värdet på high performance räknaren hämtas.
17. Tidsåtgången beräknas och lagras i minnet.
18. Värdet på high performance räknaren hämtas.
19. *free* används för att deallokera 4k bufferten.
20. Värdet på high performance räknaren hämtas.
21. Tidsåtgången beräknas och lagras i minnet.
22. Värdet på high performance räknaren hämtas.
23. *free* används för att deallokera 8k bufferten.
24. Värdet på high performance räknaren hämtas.
25. Tidsåtgången beräknas och lagras i minnet.
26. Om 10 000 värden samlats in så sparas mätvärdena till disk och programmet terminerar.
27. Gå till 2.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att allokera och deallokera de olika minnesmängderna fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

8.6.2 Allokering av delat minne

Hur lång tid tar det att allokera och deallokera delat minne?

Beskrivning:

1. Frekvensen på systemets high performance räknare hämtas via *QueryPerformanceCounter.*
2. Värdet på high performance räknaren hämtas.
3. Ett område om 2 kb delat minne skapas genom ett anrop till *CreateFileMapping((HANDLE)0xFFFFFFFF, NULL, PAGE_READWRITE, 0, 0x400, "SharedMemory").*
4. Minnet mappas in i processens minnesrymd genom ett anrop till *MapViewOfFile.*
5. Värdet på high performance räknaren hämtas.

6. Tidsåtgången beräknas och lagras i minnet.
7. Värdet på high performance räknaren hämtas.
8. Minnet tas bort ur adressrymden via ett anrop till *UnMapViewOfFile*.
9. Minnet avallokeras via *CloseHandle.*
10. Värdet på high performance räknaren hämtas.
11. Tidsåtgången beräknas och lagras i minnet.
12. Om 10 000 värden samlats in spara dessa på disk och terminera.
13. Gå till 2.

Resultat:

Samplingarna subtraheras parvis så att antalet tick som gick åt för att allokera respektive deallokera minnet fås. På dessa värden beräknas maxvärde, minvärde, medelvärde och standardavvikelse.

9 Analys

Med hjälp av dels litteraturstudien som presenterats tidigare i rapporten och dels de utförda försöken ska jag i detta kapitel visa på några egenskaper hos Windows NT.

Jag börjar med att presentera ett direkt resultat av litteraturstudien för att därefter presentera resultat från testerna.

9.1 Schemaläggning i realtidssystem

9.1.1 Antalet prioritetsnivåer bestämmer systemets storlek

Centralt i RT-sammanhang är de algoritmer för schemaläggning och hanterande av prioriteter som operativsystemet erbjuder. Prioritetsbegreppet är tex centralt i RMS.

I RMS bygger hela algoritmen på att trådarnas prioriteter bestäms av hur frekvent de ska exekveras. NT ändrar i vissa fall en tråds prioritet dynamiskt (se avsnitt 7.2.1.2: "Prioritetsnivåer ur systemets synpunkt") detta medför att RMS inte längre ger det korrekta resultatet. Det enda sättet att komma runt detta är att exekvera sitt realtidssystem i basklassen *real-time*. Denna nivå har dock endast 15 nivåer tillgängliga. Då varje tråd ska ha en egen prioritet enligt RMS så innebär detta att systemet maximalt får bestå av 15 trådar.

9.2 Metodik för analys av försöken

Då även två tillsynes lika NT-system kan vara olika om man går ned på drivrutinsnivå (olika versioner av drivrutiner, olika inställningar i registry osv) så inser man att det är i princip meningslöst att jämföra absoluta tider för de tester som genomförts. För att ändra systemets egenskaper så räcker det med att ha en drivrutin som tar längre tid på sig i avbrottshanteringen än andra versioner och därigenom ger systemet en något annorlunda fördelning mellan processortid i avbrottsrutiner och processortid till användarens processer. Mätvärdena blir även hårt knutna till den processorfamilj och klockhastighet som används i systemet.

Intressantare är då att studera hur mätvärdena varierar med olika prioriteter på testprogrammet och olika belastningar av systemet. Vilka verktyg ska då användas för att analysera resultaten?

- **Definitioner:** För att kunna definiera de olika statistiska begreppen nedan behövs följande definitioner. Antag att **n** stycken mätningar gjorts på samma storhet vid ett försök. Det j:te mätvärdet benämns då x_j där

  $\{;j \in Z, j \in 0, n]\}$

- **Medelvärde.** Definieras som $\bar{x} = \frac{1}{n}\sum_{j=1}^{n} x_j$ och är ett mått på var centrum av stickproven befinner sig.

- **Standardavvikelse.** Definieras som $s = \sqrt{\frac{1}{n-1}\sum_{j=1}^{n}(x_j - \bar{x})^2}$ och är ett mått för hur spridda stickproven är runt medelvärdet. En liten standardavvikelse betyder att stickproven är väl samlade runt medelvärdet. Vad som anses vara en liten standardavvikelse är svårt att ge någon exakt definition på, då storleksordningen på standardavikelsen beror på medelvärdets storlek.

- **Variationskoefficient.** Definieras som $d = 100\frac{s}{\bar{x}}$ alltså kvoten mellan standardavvikelse och medelvärdet uttryckt i procent. Variationskoefficienten anger hur stor standardavvikelsen är relativt medelvärdet, det är därför enklare att jämföra variationskoefficienter om man ska jämföra flera olika försök med olika medelvärden.
- **Min- och Max-värden.** Dessa anger vilket mätvärde som var minst respektive störst i mätserien.
- **1-percentilen.** Denna anger det mätvärde för vilket 1% av samtliga värden är mindre.
- **99-percentilen.** Denna anger det mätvärde för vilket 99% av samtliga värden är mindre.

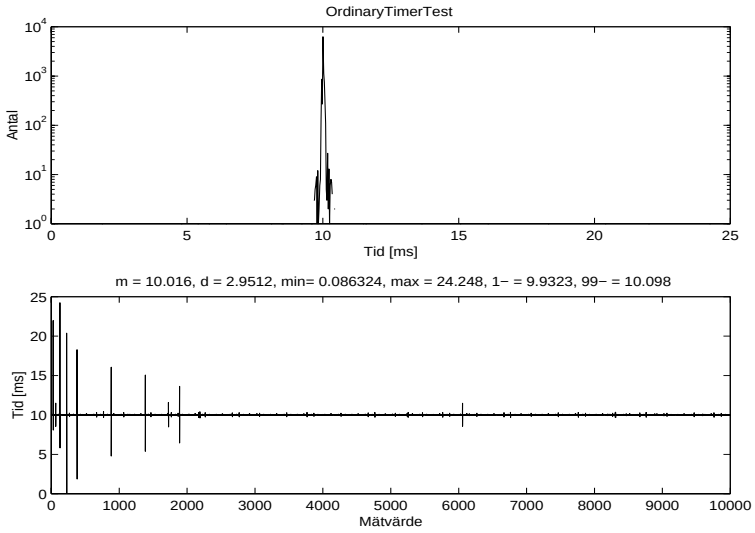
Man ska vara medveten om att de mätvärden som samlas in inte på något sätt kan tolkas som någon form av absoluta min eller max tider för en viss operation. Mätvärdena är bara samplingar ur den mängd av möjliga fördröjningar som existerar. Eventuellt kan man använda mätvärdena för att göra en skattning av vissa parametrar hos den fördelning som man antagit att mätvärdena tillhör.

9.3 Testresultat

I detta kapitel redovisas resultaten på de tester som utförts.

9.3.1 Normal timer

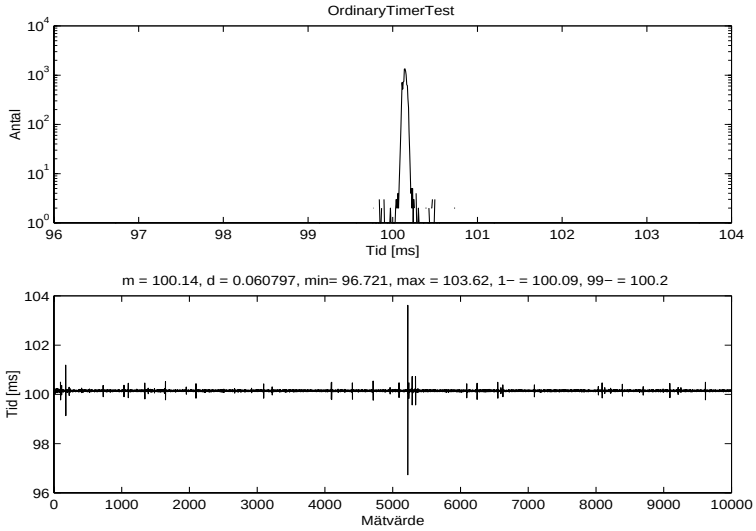
I detta stycke presenteras testresultaten för den vanliga timer-mekanismen. Dessa tester beskrivs i avsnitt 8.2.1: "Normal timer".



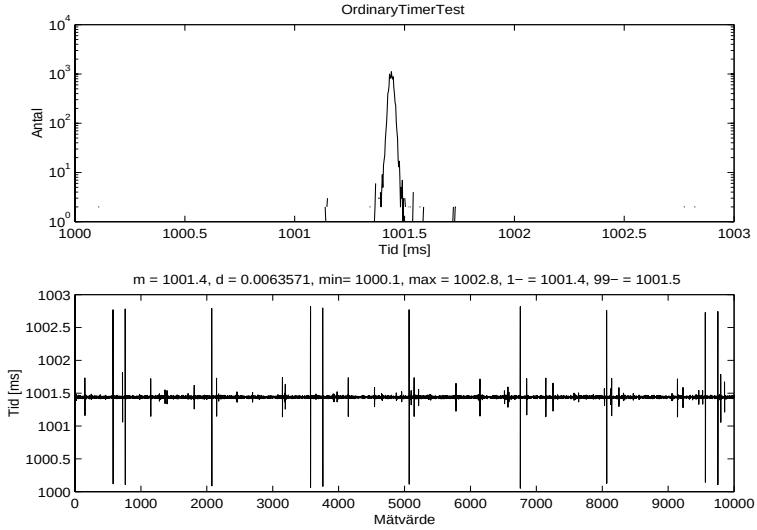
Figur 10 - Normal basprioritet obelastat system, periodtid 10 ms

Diagrammen i figur 10 presenterar mätresultatet på två olika sätt:
Underst är ett diagram som visar den uppmätta tidsåtgången som funktion av antalet mätningar.I grafen ovan ser vi alltså att tidsåtgången i regel är 10 ms vilket är förväntat men i början av mätserien förekommer både större och mindre tidsåtgång. Då många värden presenteras i grafen så kan det vara svårt att hålla isär resultat från närliggande mätningar. Ett exempel på detta ser man vid mätvärde 6000 där en mätning gett ett resultat på ca 11 ms och en mätning ca 9 ms. Dessa två mätvärden är svåra att hålla isär i figuren.
Det översta diagrammet presenterar samma mätvärden som den undre grafen. Detta diagram presenterar däremot resultatet med ett histogram. I detta diagram ligger tidsåtgången för försöket efter x-axeln och antalet mätningar med ett visst värde kan ses på y-axeln som har logaritmisk skala. I diagrammet ovan ser man att flertalet värden ligger på 10ms, ett antal värden ligger dock lite skilda från 10ms och ger upphov till att histogrammet påminner om en spetsig triangel.

Denn typ av diagram kommer genomgående att användas i detta kapitel för att presentera resultaten av testerna.



Figur 11 - Normal basprioritet obelastat system, periodtid 100 ms

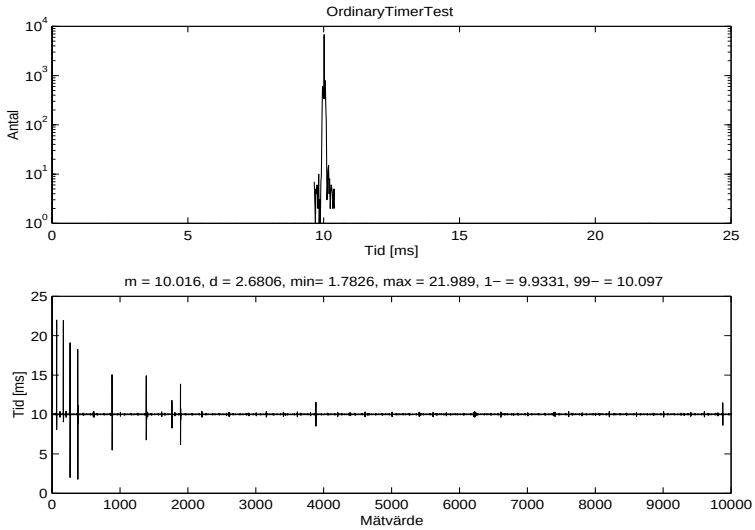


Figur 12 - Normal basprioritet obelastat system, periodtid 1000 ms

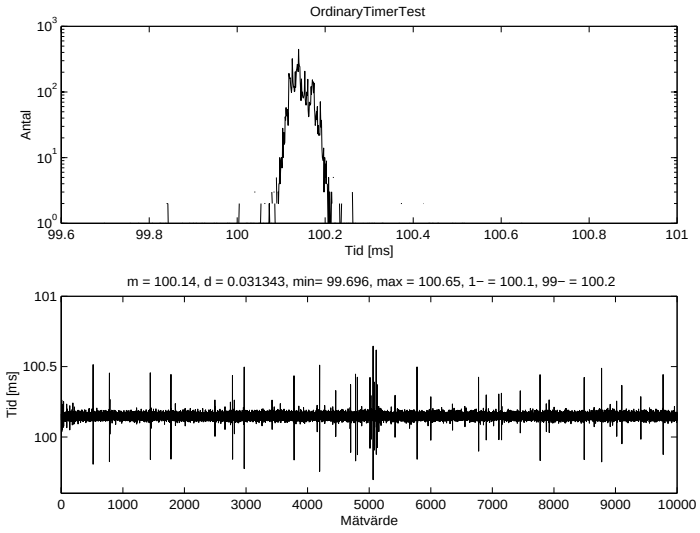
Kommentarer: figur 10, 11 och 12

I fallet med obelastat system och normal basprioritet verkar det som om man i stort sett kan lita på den normala timer-mekanismen. Fördröjningarna avviker något från de ideala värdena men det är svårt att säga om detta beror på den mätmetod som använts eller Microsofts implementation. Man kan dock i 10 ms fal-

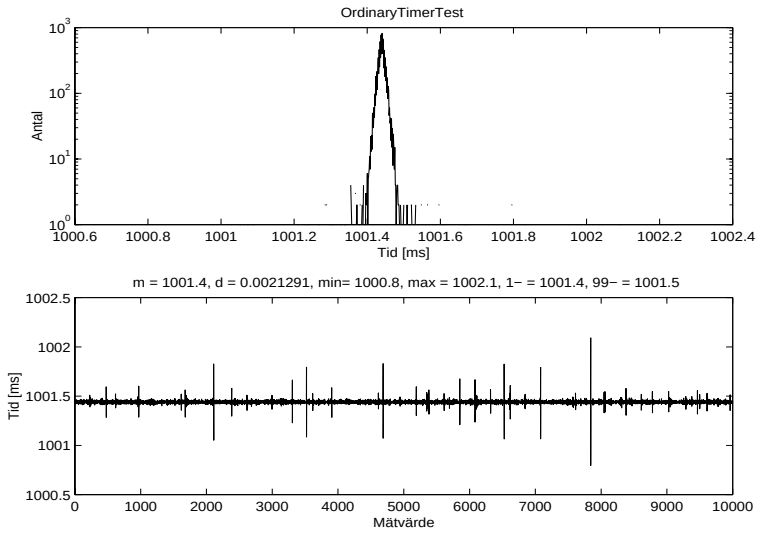
let se ett antal spikar då fördröjningen avvikit mycket från de ideala 10 millisekunderna..



Figur 13 - Hög basprioritet obelastat system, periodtid 10 ms



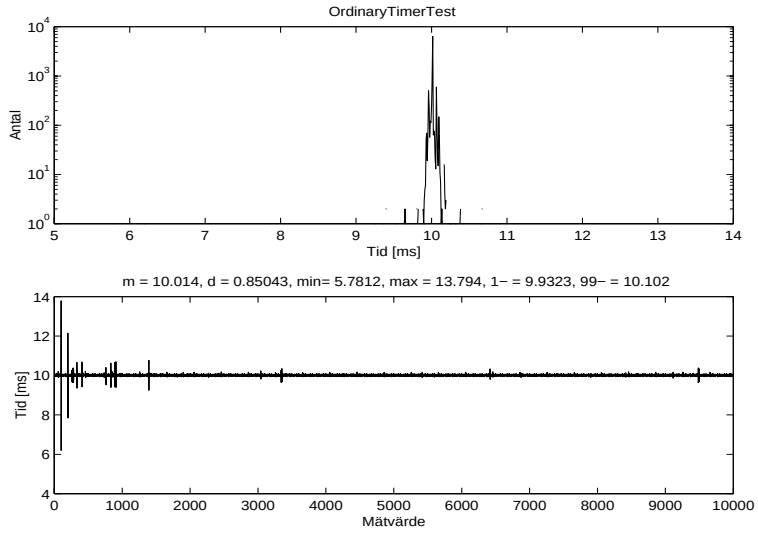
Figur 14 - Hög basprioritet obelastat system, periodtid 100 ms



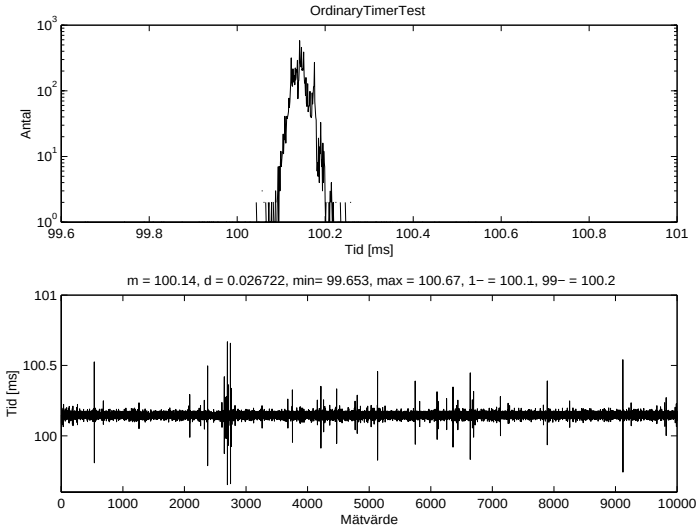
Figur 15 - Hög basprioritet obelastat system, periodtid 1000 ms

Kommentarer: figur 13, 14 och 15

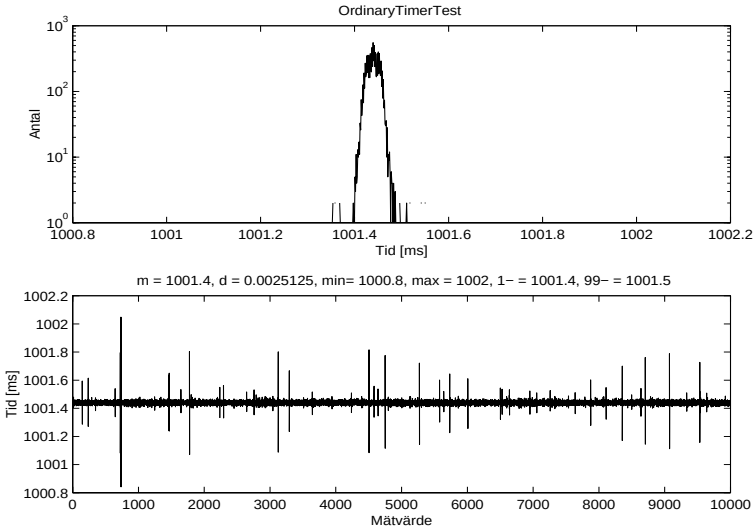
I fallet med obelastat system och hög basprioritet ser resultatet i stort sätt ut som i fallet med normal basprioritet. I 10 ms fallet finns fortfarande ett antal spikar men i de andra två fallen är spridningen mindre än i fallet med normal basprioritet.



Figur 16 - Realtids basprioritet obelastat system, periodtid 10 ms



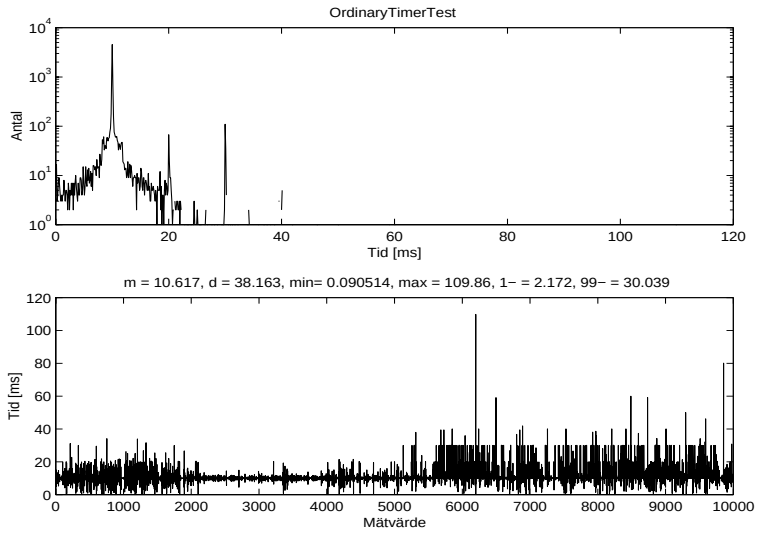
Figur 17 - Realtids basprioritet obelastat system, periodtid 100 ms



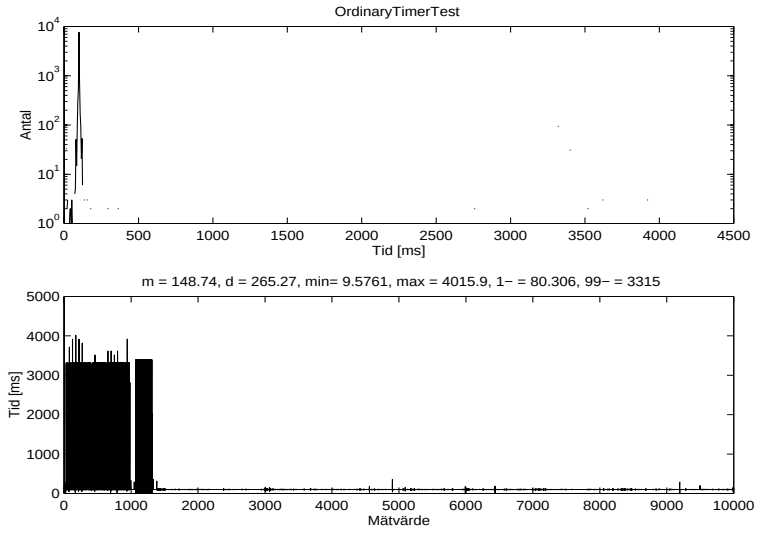
Figur 18 - Realtids basprioritet obelastat system, periodtid 1000 ms

Kommentarer: figur 16, 17 och 18

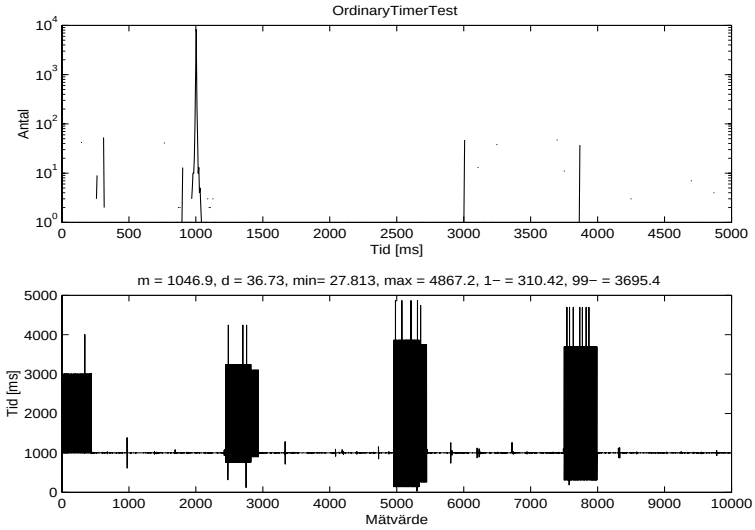
I fallet med obelastat system och realtids basprioritet är resultatet för 10 ms periodtid betydligt bättre än i de tidigare fallen. Variationskoefficienten d är nu ca 0.85, där den i fallet med normal och hög basprioritet varit ca 2.9. För övriga fall har variationskoefficienten minskat något.



Figur 19 - Normal basprioritet belastat system, periodtid 10 ms



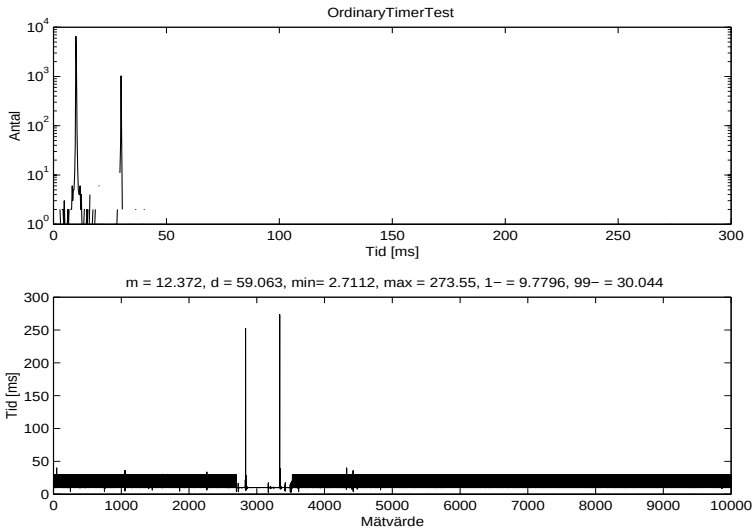
Figur 20 - Normal basprioritet belastat system, periodtid 100 ms



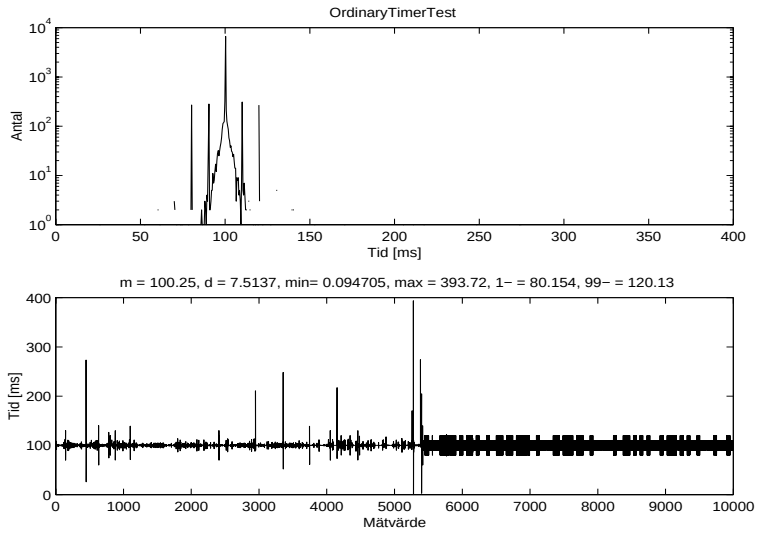
Figur 21 - Normal basprioritet belastat system, periodtid 1000 ms

Kommentarer: figur 19, 20 och 21

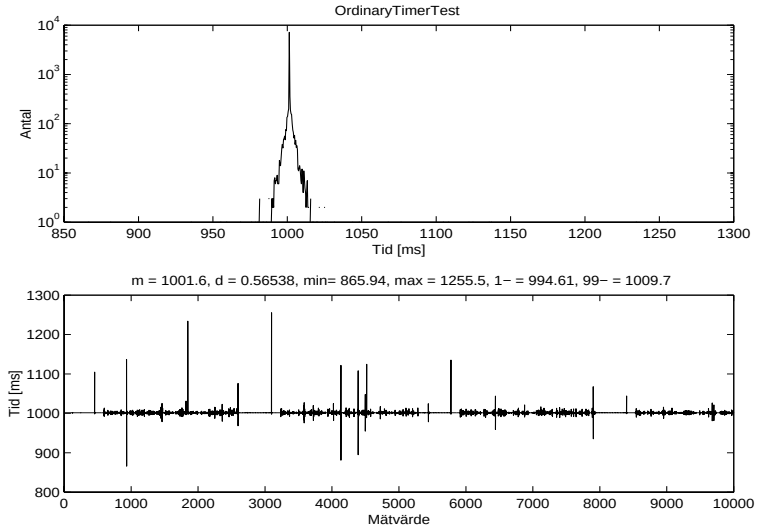
När systemet belastas och lasten har samma basprioritet som testprogrammet så syns det tydligt i diagrammen. För samtliga testfall finns block av mätningar där den uppmätta tiden är betydligt större än det förväntade värdet.



Figur 22 - Hög basprioritet belastat system, periodtid 10 ms



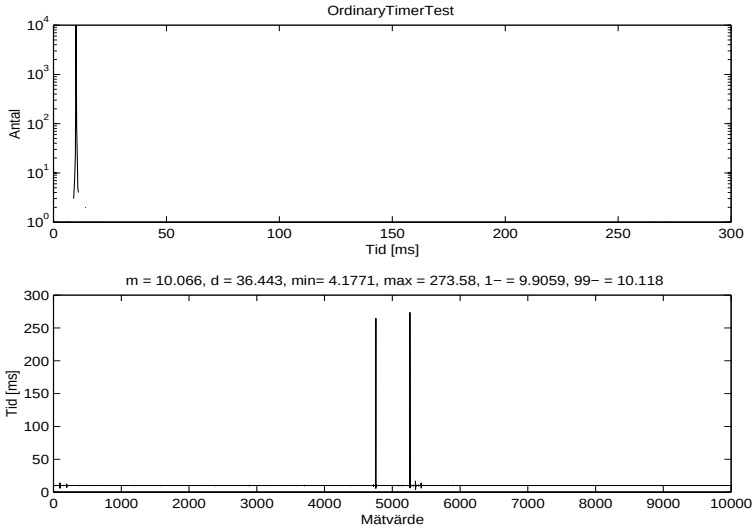
Figur 23 - Hög basprioritet belastat system, periodtid 100 ms



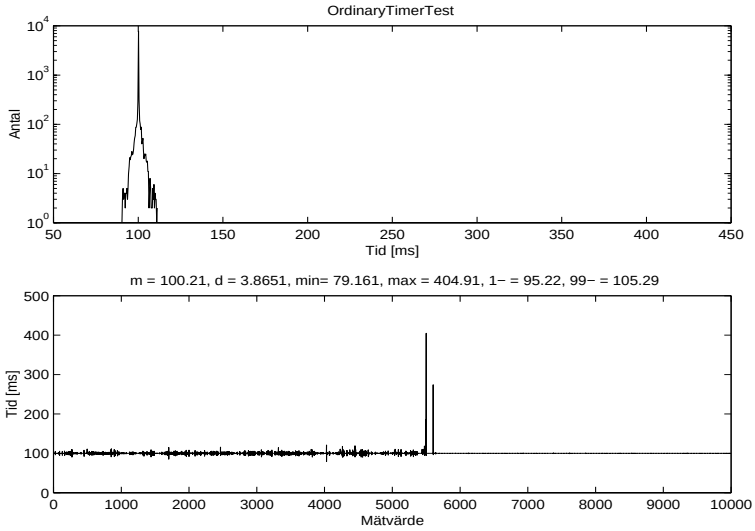
Figur 24 - Hög basprioritet belastat system, periodtid 1000 ms

Kommentarer: figur 22, 23 och 24

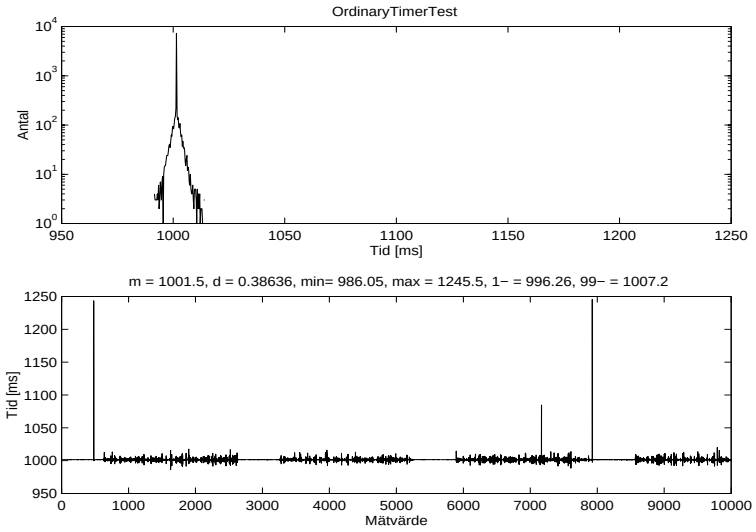
När testprogrammet får högre prioritet än lasten minskar lastens inverkan betydligt men en hel del spikar förekommer fortfarande. Dessa beror antagligen på hårddiskanvändning där interupthanteringsrutinerna blockerar det övriga systemet.



Figur 25 - Realtids basprioritet belastat system, periodtid 10 ms



Figur 26 - Realtids basprioritet belastat system, periodtid 100 ms



Figur 27 - Realtids basprioritet belastat system, periodtid 1000 ms

Kommentarer: figur 25, 26 och 27

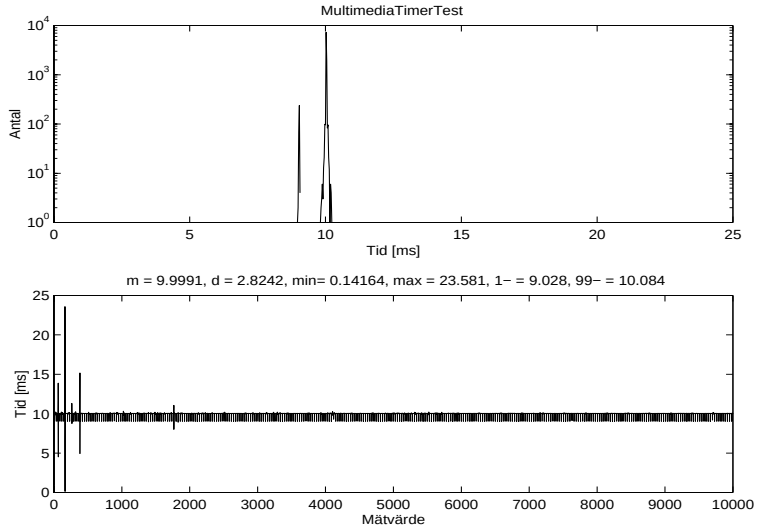
I fallet med realtids basprioritet på testprogrammet blir resultaten ännu bättre men det existerar fortfarande större avvikelser från de förväntade värdena.

9.3.1.1 Multimediatimers

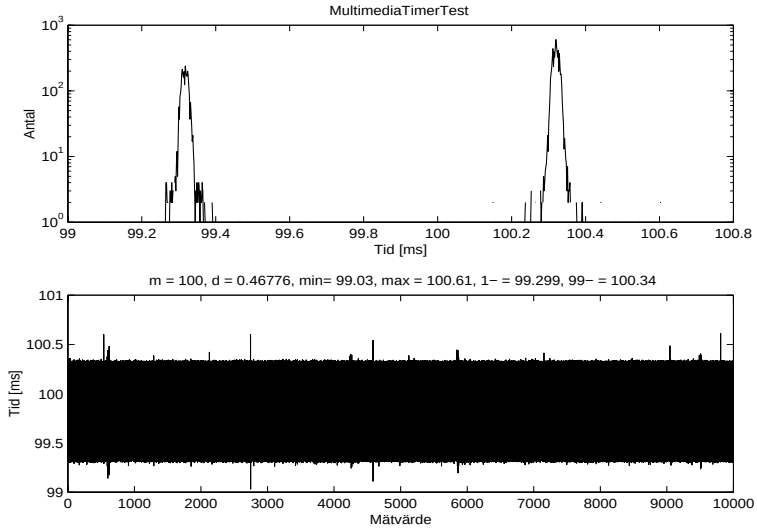
Detta stycke beskriver resultaten av de tester som beskrivs i avsnitt 8.2.2: "Multimedia Timers".

Medvetna om att den vanliga timermekanismen inte alltid mötte de krav som ställs vid multimediatillämpningar tog man fram en ny förbättrad mekanism. Bland annat använder man callback-funktioner istället för meddelanden för att signalera att timern löst ut.

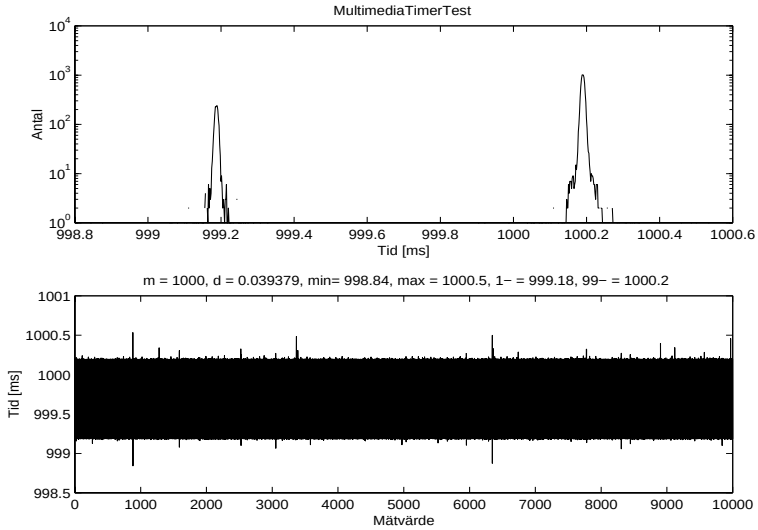
Hur ser då resultaten för multimedia-timers ut?



Figur 28 - Normal basprioritet obelastat system, periodtid 10 ms



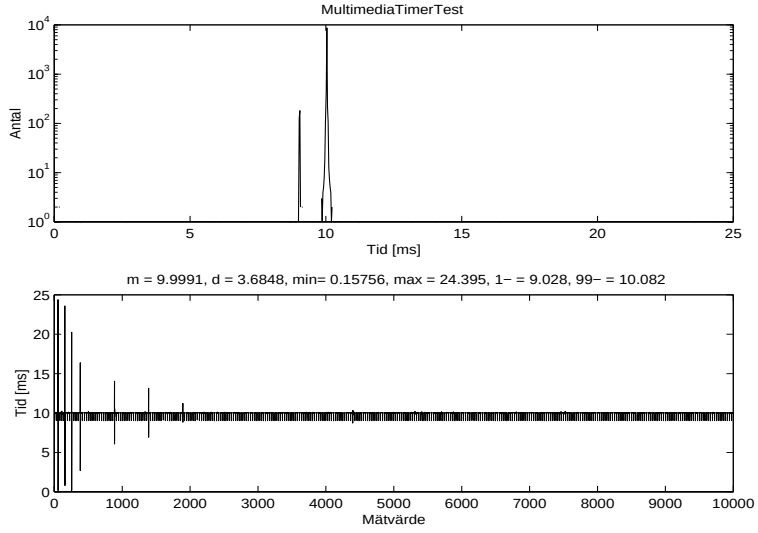
Figur 29 - Normal basprioritet obelastat system, periodtid 100 ms



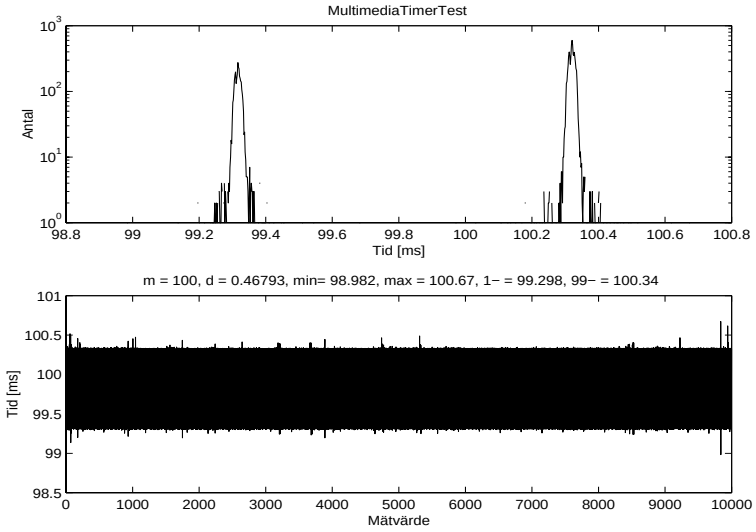
Figur 30 - Normal basprioritet obelastat system, periodtid 1000 ms

Kommentarer: figur 28, 29 och 30

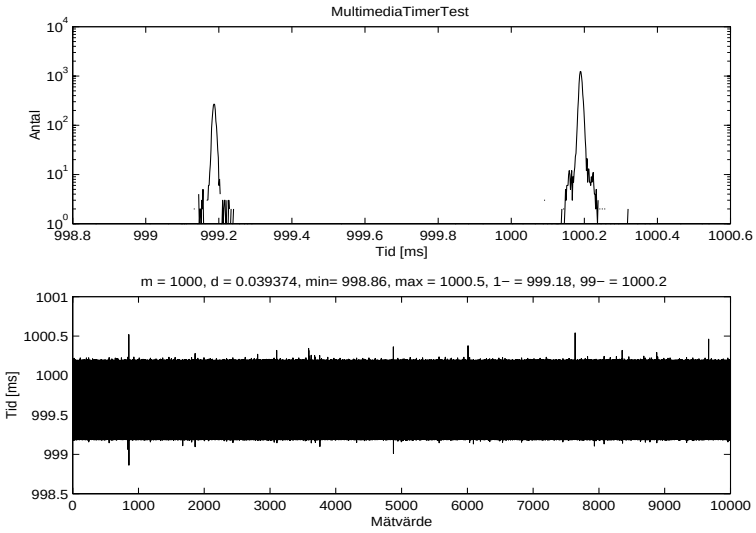
I fallet med normal basprioritet och obelastat system är det ingen större skillnad mellan multimediatimers och vanliga. De har i stort sätt bra egenskaper men ett fåtal spikar existerar.



Figur 31 - Hög basprioritet obelastat system, periodtid 10 ms



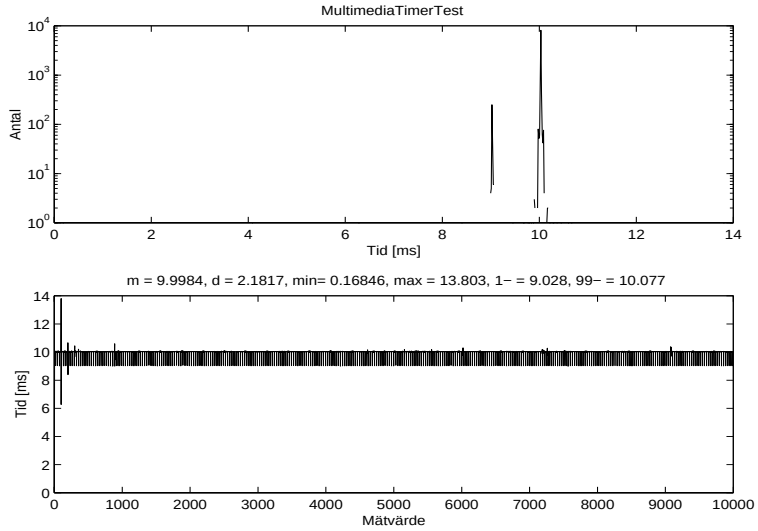
Figur 32 - Hög basprioritet obelastat system, periodtid 100 ms



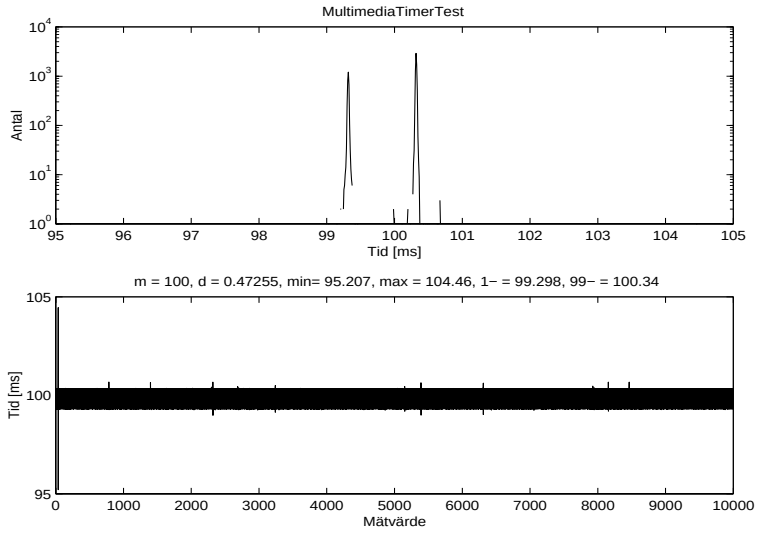
Figur 33 - Hög basprioritet obelastat system, periodtid 1000 ms

Kommentarer: figur 31, 32 och 33

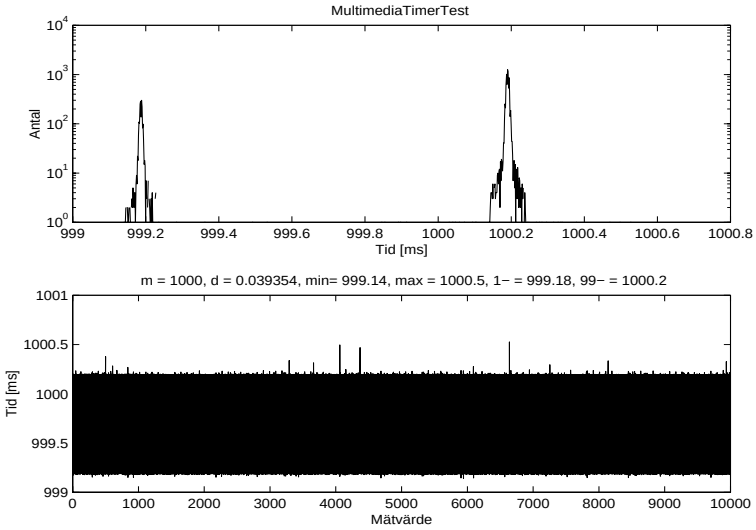
Inte heller i detta fall är det några större skillnader jämfört med den normala timermekanismen, förrutom att variationskoefficienten är något större med multimediatimers vilket är fövånande då dessa påstås vara bättre timers än de vanliga.



Figur 34 - Realtids basprioritet obelastat system, periodtid 10 ms



Figur 35 - Realtids basprioritet obelastat system, periodtid 100 ms

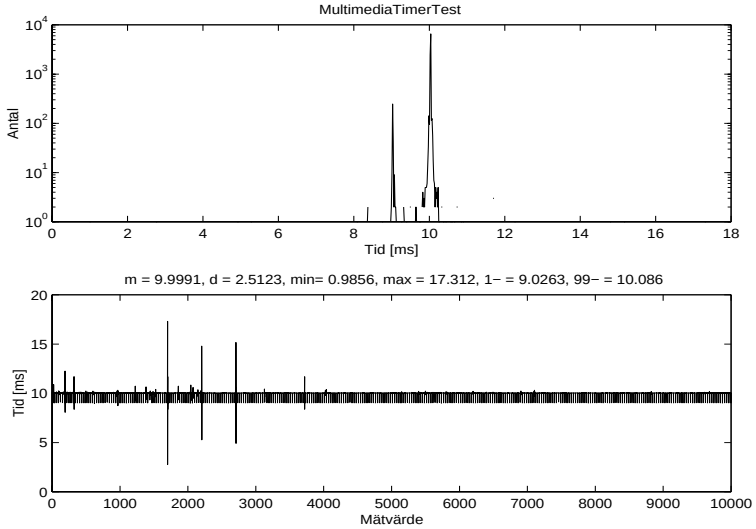


Figur 36 - Realtids basprioritet obelastat system, periodtid 1000 ms

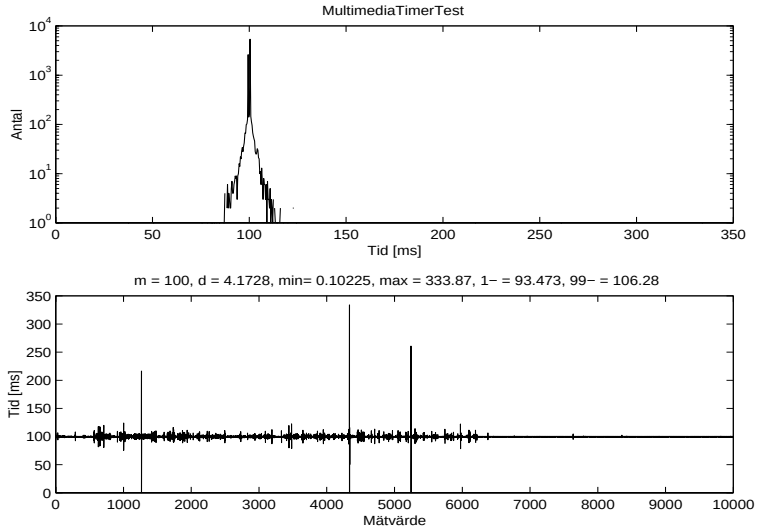
Kommentarer: figur 34, 35 och 36

Detta fall har ett par intressanta egenskaper. Medelvärdet på fördröjningarna är bättre än i fallet med normala timers. Vid fördröjningar med 100 ms och 1000 ms är medelvärdena exakt 100 ms respektive 1000 ms, men variationskoefficienten är större än i fallet med normala timers.

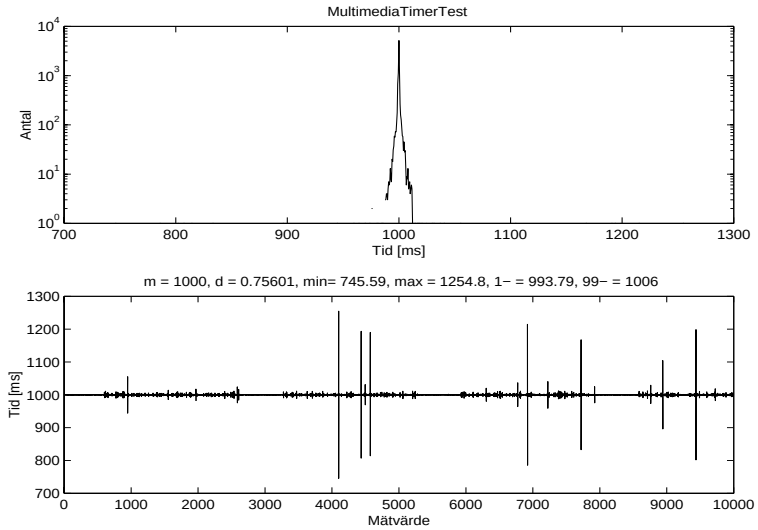
Vad händer då när systemet belastas?



Figur 37 - Normal basprioritet belastat system, periodtid 10 ms



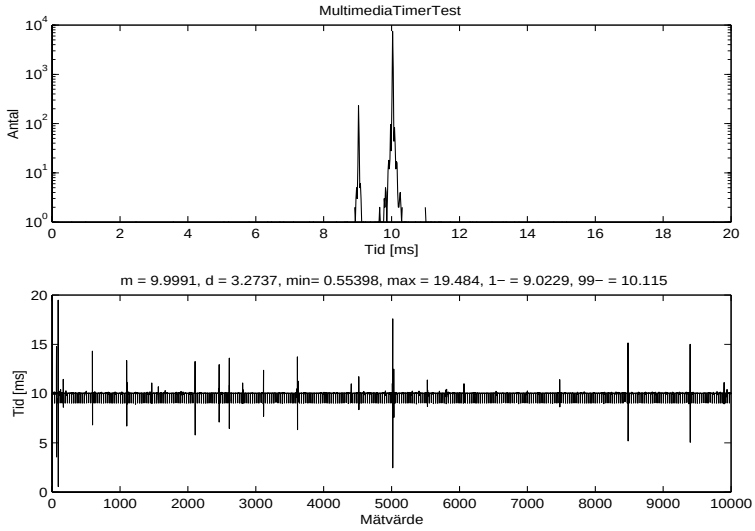
Figur 38 - Normal basprioritet belastat system, periodtid 100 ms



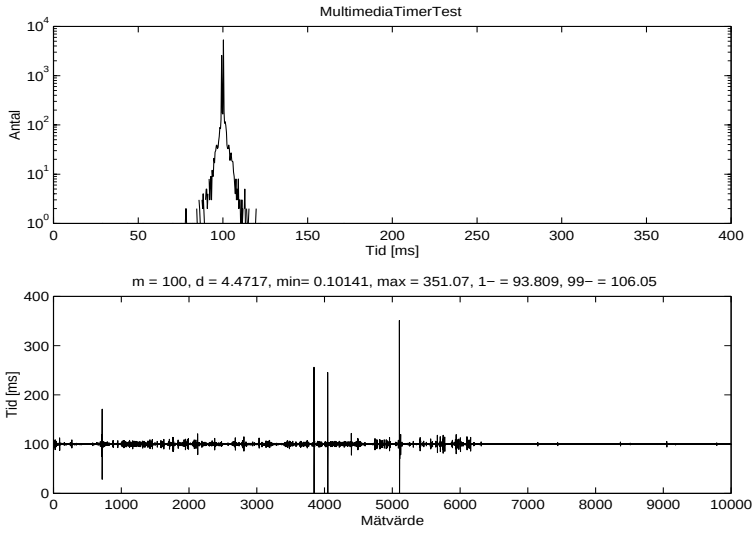
Figur 39 - Normal basprioritet belastat system, periodtid 1000 ms

Kommentarer: figur 37, 38 och 39

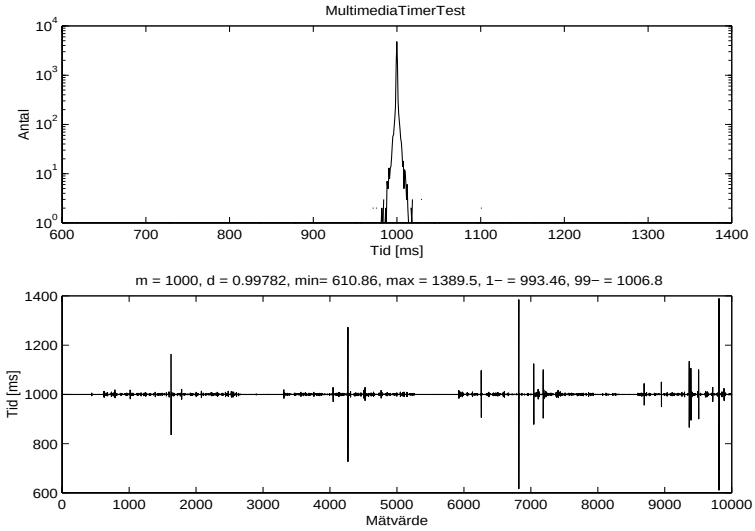
Detta test visar att belastningen inte nämnvärt påverkar timerns uppträdande. Spridningen är lite större än i det obelastade fallet. Om en jämförelse med uppträdandet hos den vanliga timern vid belastning görs är skillnaden mycket stor.



Figur 40 - Hög basprioritet belastat system, periodtid 10 ms



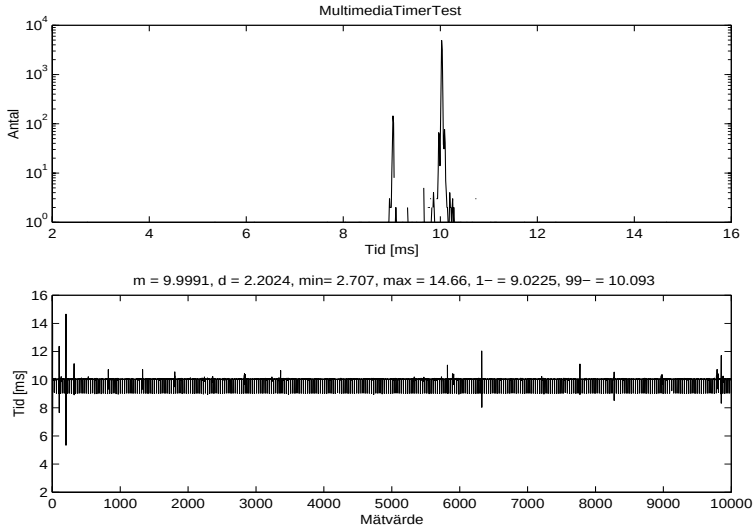
Figur 41 - Hög basprioritet belastat system, periodtid 100 ms



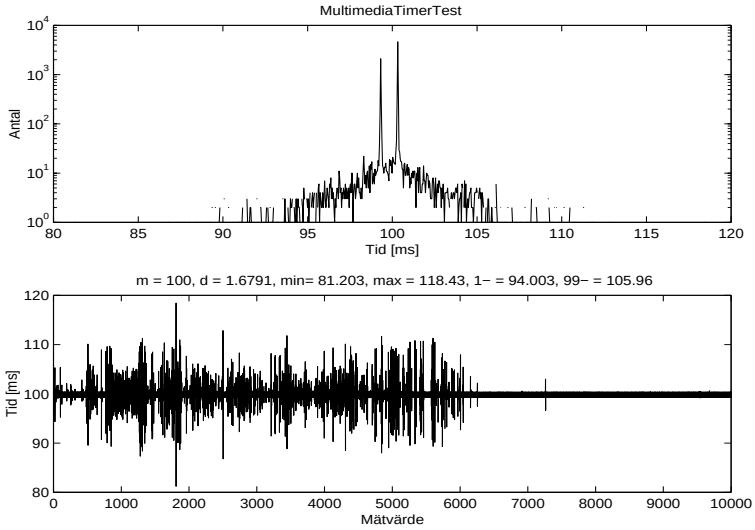
Figur 42 - Hög basprioritet belastat system, periodtid 1000 ms

Kommentarer: figur 40, 41 och 42

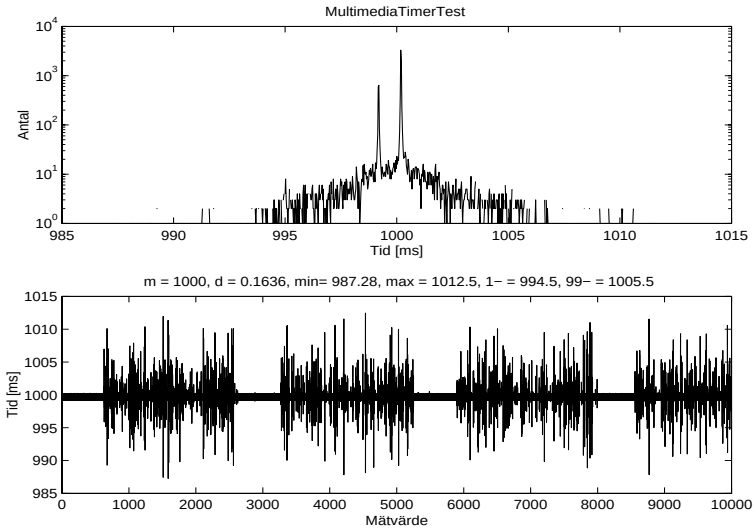
Det är ingen större skillnad mellan hög och normal basprioritet i detta fall.



Figur 43 - Realtids basprioritet belastat system, periodtid 10 ms



Figur 44 - Realtids basprioritet belastat system, periodtid 100 ms



Figur 45 - Realtids basprioritet belastat system, periodtid 1000 ms

Kommentarer: figur 43, 44 och 45

Realtids basprioritet innebär inte heller det någon större skillnad mot de tidigare fallen.

Jämför man multimediatimers och vanliga timers kan man notera följande skillnader:

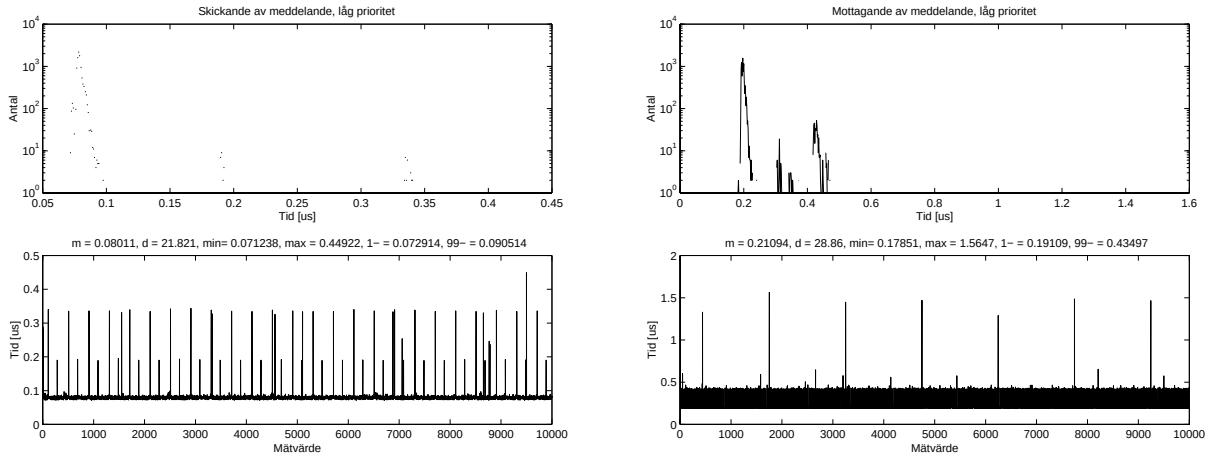
- **Känslighet för belastning:** Vanliga timers är mycket känsliga för belastning, om systemet belastas har fördröjningar på ca 3 sekunder observerats i fallen med normal basprioritet 100 ms och 1000 ms. Multimedia timers däremot är relativt okänsliga för belastning.
- **Beroende av basprioritet**: Vanliga timers uppträder även olika beroende på vilken basprioritet som det anropande programmet har. När systemet belastas ser man en klar skillnad mellan olika basprioriteter. Denna skillnad beror antagligen på att meddelanden används av den normala timern för kommunikation med den användande processen. Man vet inte på förhand hur lång tid det tar innan den anropade processen får tillfälle att ta hand om meddelandena i meddelandekön nästa gång. Multimediatimers använder sig däremot av callback-funktioner, vilket ger kvickare respons.

Skillnaderna beror till största delen på att multimediatimers använder en tråd på hög prioritet för tidsmätningen och anropandet av callbackfunktioner. Det enda sättet att störa en multimediatimer är genom trådar med högre eller samma prioritet än timertråden eller med interupt.

9.3.1.2 Tid för att skicka ett meddelande

I detta stycke beskrivs resultaten av testerna av meddelandehanteringen. Dessa tester beskrivs i styckena 8.3.1, 8.3.2 och 8.3.3.

Till en början låter vi den mottagande tråden ha en lägre prioritet än den sändande, dvs den mottagande tråden startas ej förrän den sändande tråden lägger sig och väntar.

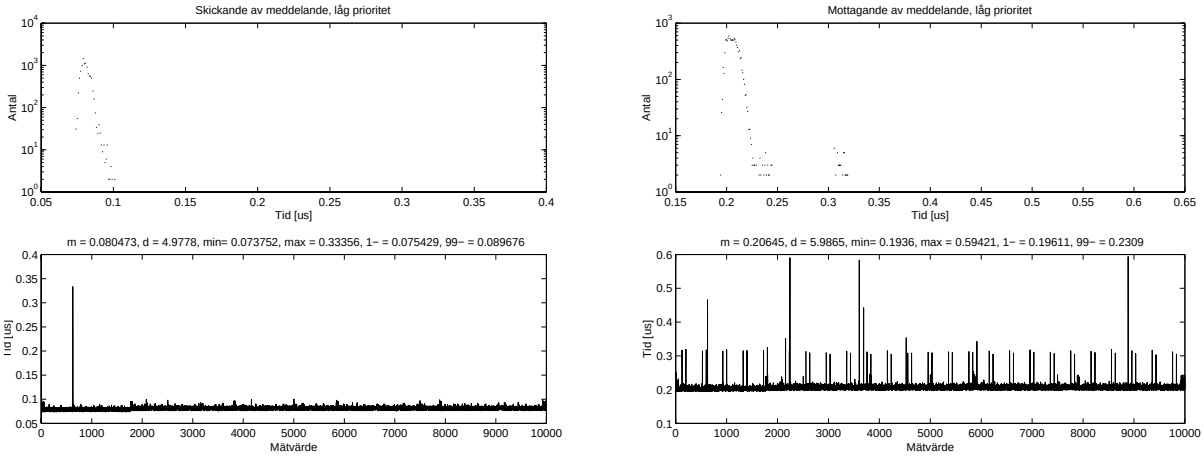


Figur 46 - Normal basprioritet, obelastat system, mottagande tråd har lägre prioritet än den sändande

Kommentarer: figur 46

Även om systemet är obelastat så ser man att någon periodisk aktivitet förekommer i systemet. På sändsidan har topparna en höjd som är nästan 4 gånger så stor

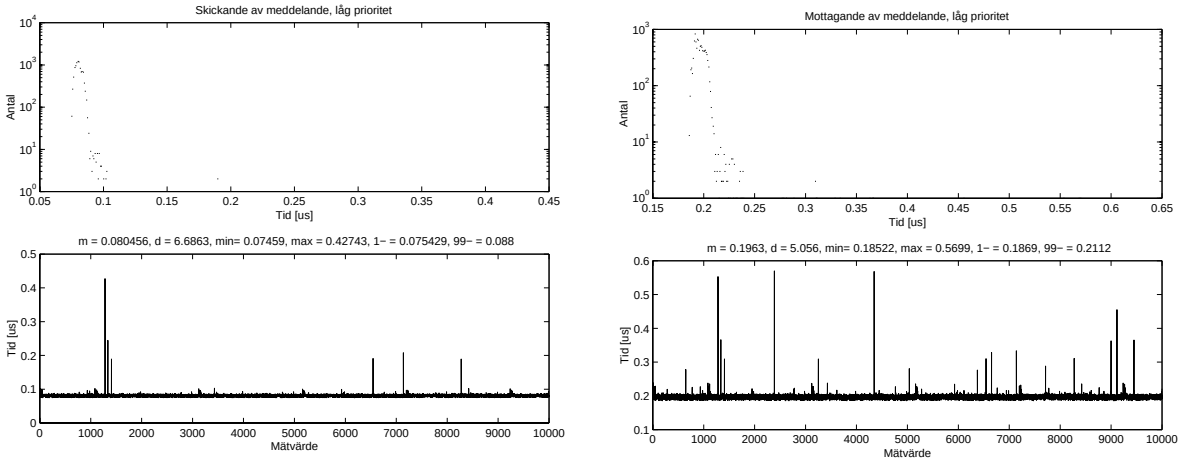
som medelvärdet och på mottagarsidan är topparna ca 8 gånger så stora som medelvärdet.



Figur 47 - Hög basprioritet, obelastat system, mottagande tråd har lägre prioritet än den sändande

Kommentarer: figur 47

När basprioriteten höjs så minskar antalet toppar som avviker från medelvärdet.



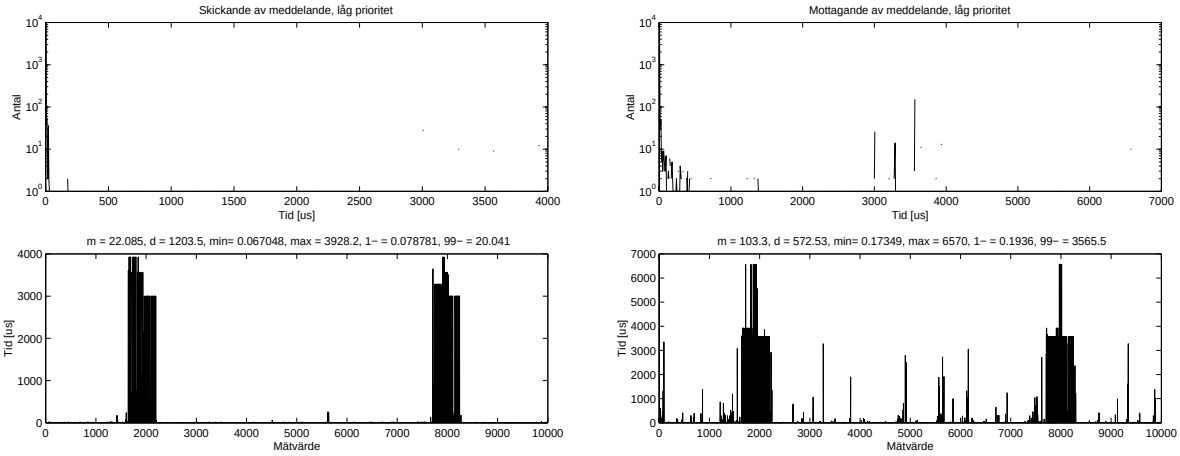
Figur 48 - Realtids basprioritet, obelastat system, mottagande tråd har lägre prioritet än den sändande

Kommentarer: figur 48

När prioriteten höjs ännu mer upp till realtidsområdet så blir det ingen större skillnad i uppförande. Detta är inte särskilt överaskande så systemet inte är nedlastat och relativt få processer ska samsas om de tillgängliga systemresurserna.

I alla tre fallen så är medelvärdena för operationerna i stort sett lika. Detta värde befinner sig antagligen väldigt nära det minsta värde som praktiskt går att uppnå.

Vad händer nu när vi belastar systemet?

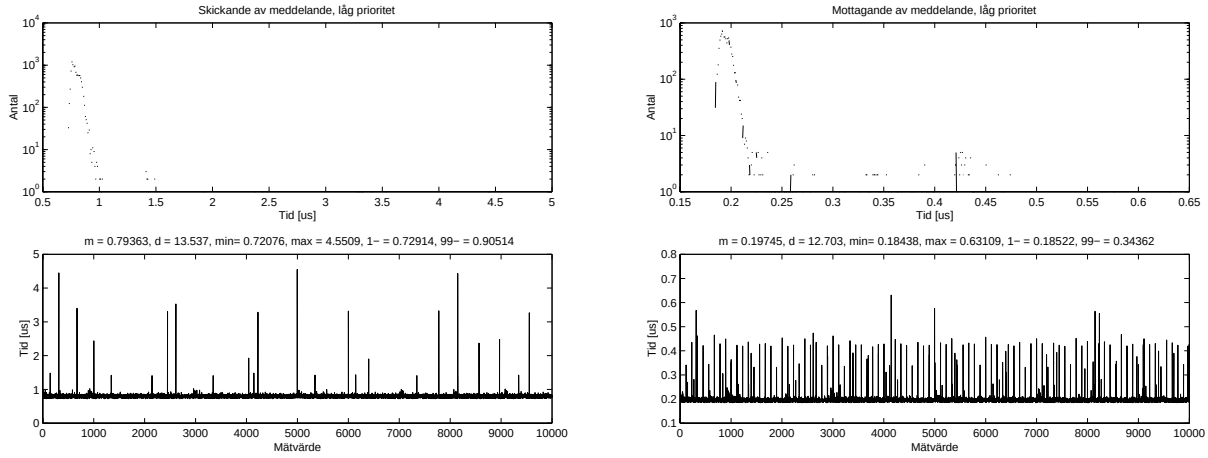


Figur 49 - Normal basprioritet, belastat system, mottagande tråd har lägre prioritet än den sändande

Kommentarer: figur 49

När systemet belastats ser man att belastningen ger en maximal fördröjning som är mer än 1000 gånger större än i det obelastade fallet.

I det obelastade fallet beror topparna på att det finns systemprocesser med en högre prioritetsnivå än testprogrammet. I det belastade fallet så existerar det desutom en process på samma prioritetsnivå som kraftigt belastar systemet.

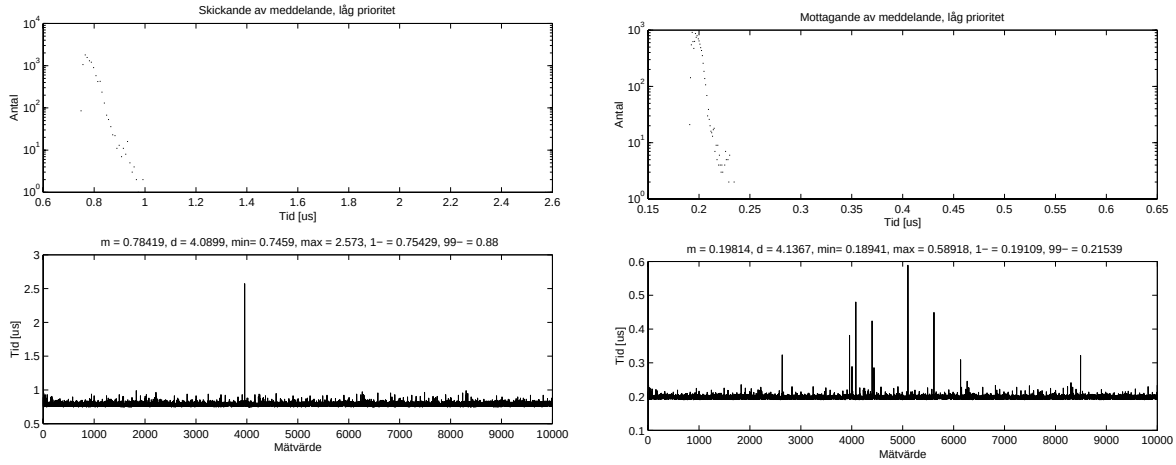


Figur 50 - Hög basprioritet, belastat system, mottagande tråd har lägre prioritet än den sändande

Kommentarer: figur 50

Även i detta fall så förekommer toppar med betydligt större fördröjningar än normalt. Då flera systemprocesser har samma prioritet som testprocessen så avbryter de varandra och större fördröjningar kan uppstå. Jämför man variationskoefficientens värde för normal basprioritet obelastad 21.82 och hög basprioritet obelastad 4.98 ser man att skillnaden är en faktor 4.

När systemet belastas så ser vi att svarstiderna ökar och att de är ca 10 gånger högre än i det obelastade fallet. Men om man jämför detta med fallet normal basprioritet belastat system så är topparna ca 1000 gånger mindre och medelvärdet 20 gånger mindre.



Figur 51 - Realtids basprioritet, belastat system, mottagande tråd har lägre prioritet än den sändande

Kommentarer: figur 51

Nu befinner vi oss på realtidsnivåerna i schemaläggningen. Testprocessens prioritet är nu högre än många av systemprocesserna. Däremot finns det några systemprocesser som exekverar på realtids basprioritet. Bland dessa kan nämnas minneshanteraren, cache-hanteraren och vissa device drivers. Testprogrammet tävlar med dessa om processortid och vissa toppar existerar men de är mindre både till storlek och antal än för testfallen med lägre prioritet.

Även på denna prioritetsnivå är de absoluta värden på tiden som gått åt för att skicka ett meddelade ca 10 gånger högre än för det obelastade fallet. Om man bortser från de absoluta värdena utan istället studerar variationskoefficienten så finner man relativt få värden som avviker från de övriga. I just dessa tester så blev det så att variationskoefficienten var mindre för det belastade fallet än i det obelastade.

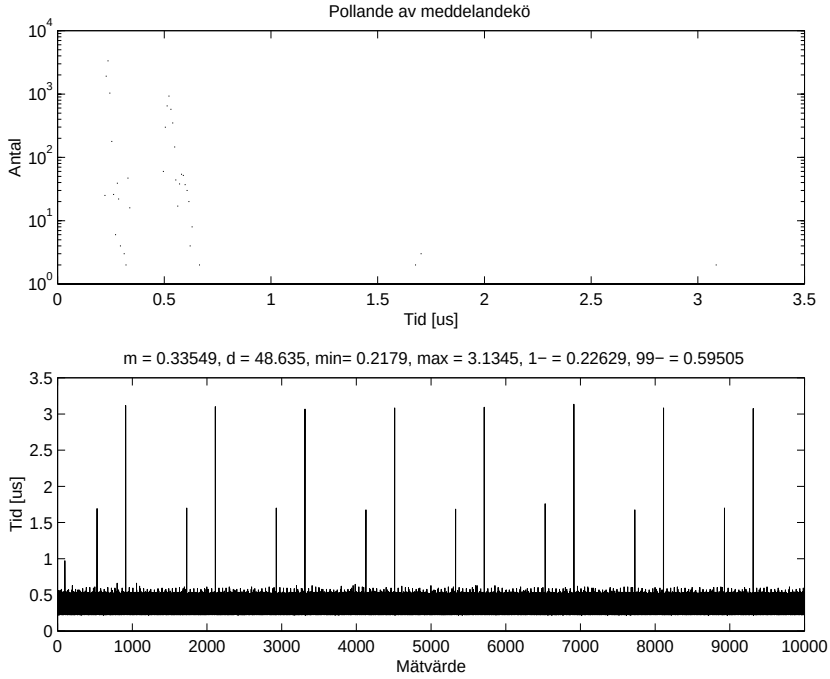
Dessa tester utfördes även med andra prioriteter på den mottagande tråden. De fall som undersöktes var när den mottagande tråden hade samma samt högre prioritet än den sändande. Resultaten från dessa försök avvek inte nämnvärt från de

resultat som presenterats här, graferna finns därför i avsnitt 12.1.1: "Tid för att skicka ett meddelande".

9.3.1.3 Tid för att kontrollera om något meddelande finns i kö

I detta stycke beskrivs resulteten av testet beskrivet i avsnitt 8.3.4: "Tid för att kontrollera om något meddelande finns i kö"

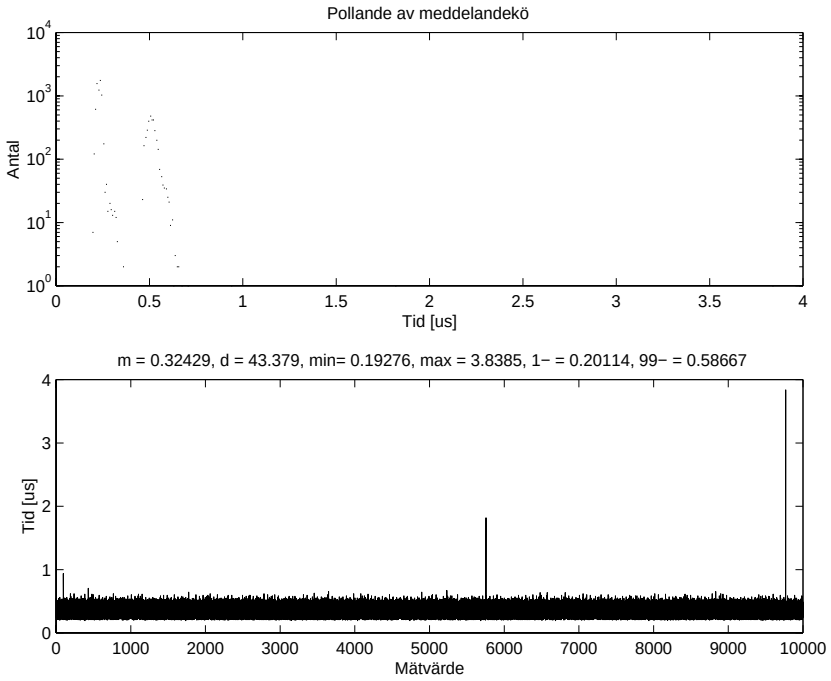
I detta test mättes hur lång tid det tog att kontrollera om något meddelande fanns i meddelandekön.



Figur 52 - Normal basprioritet obelastat system

Kommentarer: figur 52

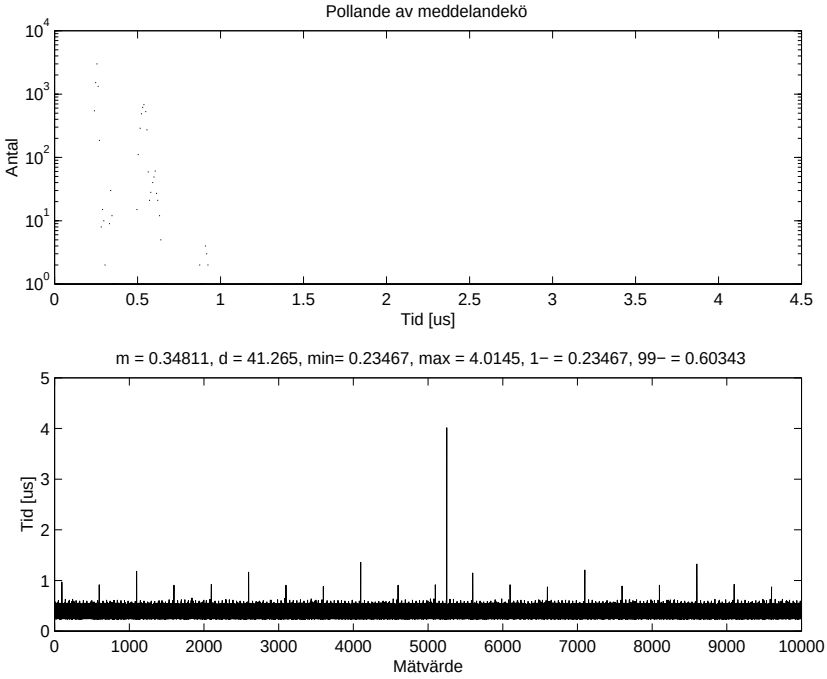
Ur denna graf kan man utläsa att det i allmänhet tar ca 0.3 ms att kontrollera om något meddelande finns i kön för inkommande meddelanden. Ett antal tillfällen då operationen tagit ca 1.7 ms förekommer. Ett antal toppar på ca 3 ms förekommer även. Dessa toppar återkommer i ett regelbundet mönster under hela försöket.



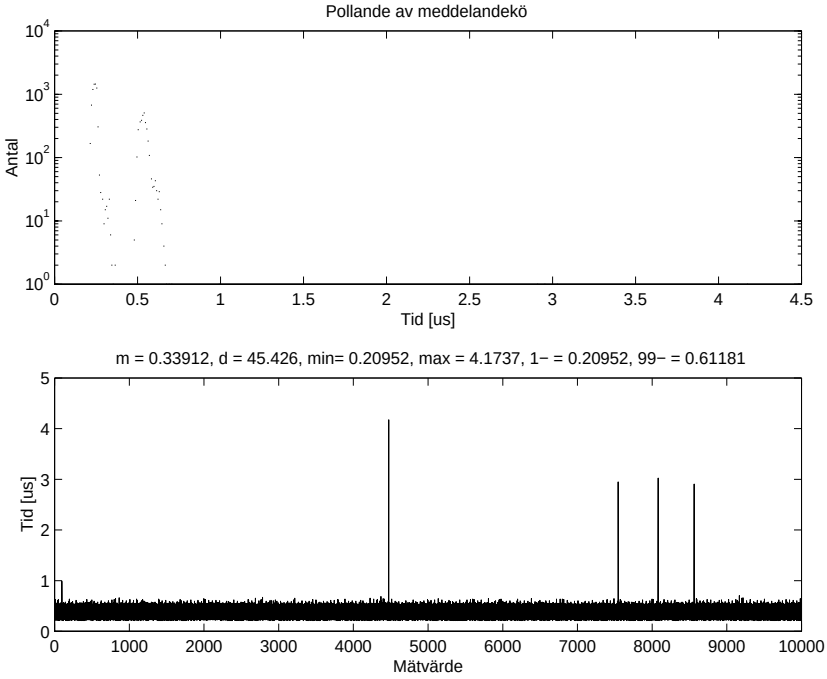
Figur 53 - Normal basprioritet belastat system

Kommentarer: figur 53

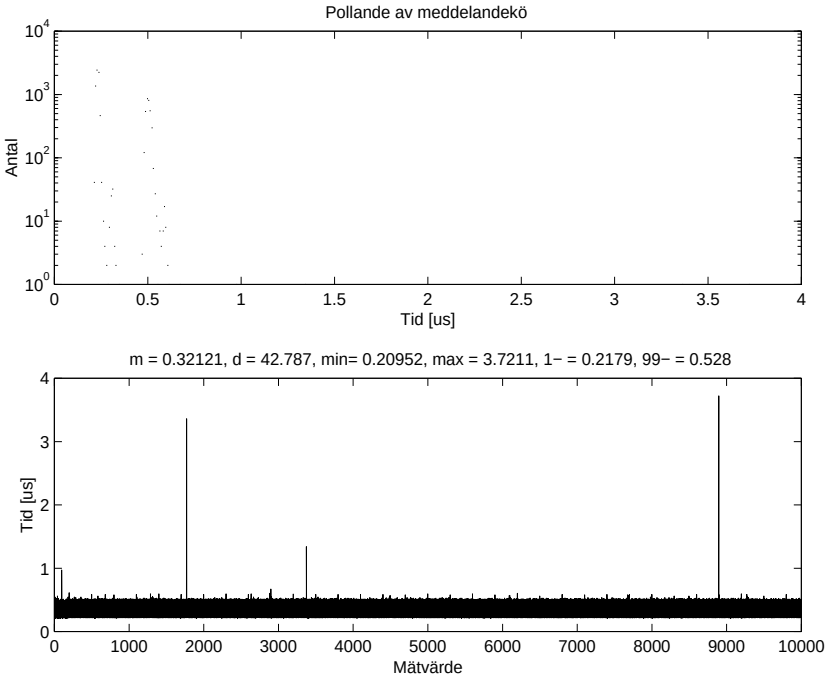
När systemet belastas så sker ingen större förändring av vare sig medelvärde eller variationskoefficient. Värt att notera är att de cykliska topparna som kunde ses i figur 52 inte förekommer i detta försök. Detta trots att systemet nu är belastat och borde ha sämre egenskaper än tidigare.



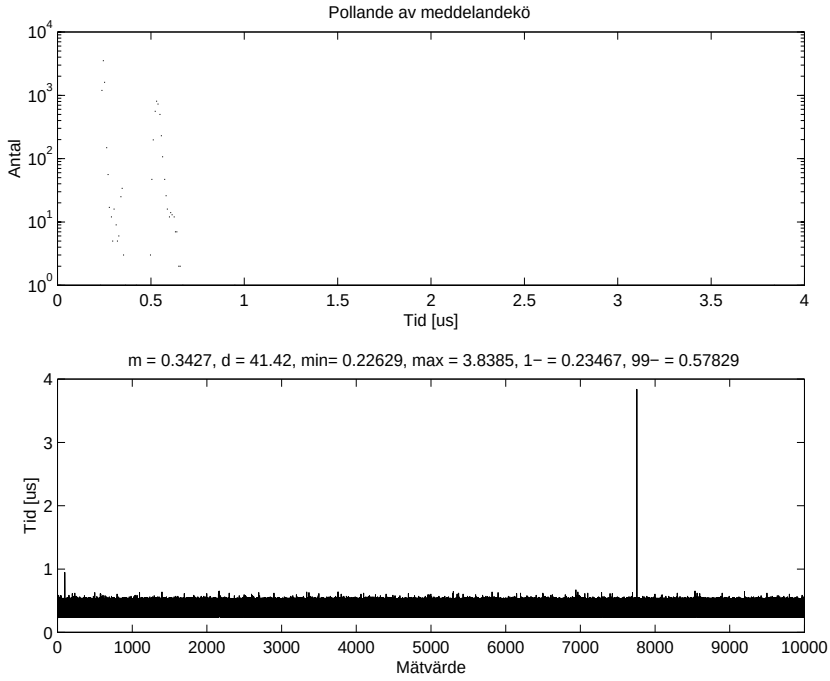
Figur 54 - Hög basprioritet obelastat system



Figur 55 - Hög basprioritet belastat system



Figur 56 - Realtids basprioritet obelastat system



Figur 57 - Realtids basprioritet belastat system

Kommentarer: figur 54, 55, 56 och 57

Trots att basprioriteterna förändras sker ingen större förändring av medelvärdet och variationskoefficienten. Att kontrollera meddelandekön tar lite mer än 0.33 µs i samtliga fall.

Precis som i tidigare försök så ser vi några få tillfällen då fördröjningen är mångdubbelt längre än vanligt.

Resultatet av detta försök är att oberoende av belastning och prioritetsnivå så tar det i princip alltid lika lång tid att kontrollera om något meddelande finns i systemets meddelandekö. Detta är ett tecken på att operativsystemet inte har någon form prioriteringsmekanism vid meddelandehantering. Alla som vill titta i meddelandekön behandlas på samma sätt oavsett prioritet. Det cykliska utseendet på topparna i figur 52 uppträder inte i några andra försök och härstammar antagligen från en yttre källa.

9.3.1.4 Eventhantering

En annan metod för interprocess kommunikation är händelser. De försök som beskrivs i avsnitt 8.3: "Beskrivning av tester för meddelandehantering" utfördes men med händelser istället för meddelanden (försöken beskrivs i avsnitt 8.4: "Eventhantering"). Resultaten av detta försök var principiellt samma som i 9.3.1.2. Graferna presenteras därför inte här utan återfinns i avsnitt 12.1.2: "Eventhantering".

I tabell 6 och tabell 7 ses en jämförelse av testresultaten för fallen där den mottagande tråden har lägre prioritet än den sändande. Både det obelastade och det belastade fallet redovisas.

IPC-Mekanism	Basprioritet	Medeltid för sändning [µs]	Medeltid för mottagande [µs]
Händelser	Låg	0.34	0.07
Meddelande	Låg	0.08	0.21
Händelser	Hög	0.34	0.82
Meddelande	Hög	0.08	0.20
Händelser	Realtid	0.32	0.08
Meddelande	Realtid	0.08	0.20

Tabell 6 - Jämförelse av tidsåtgång vid meddelandehantering med händelser och meddelanden, obelastat system.

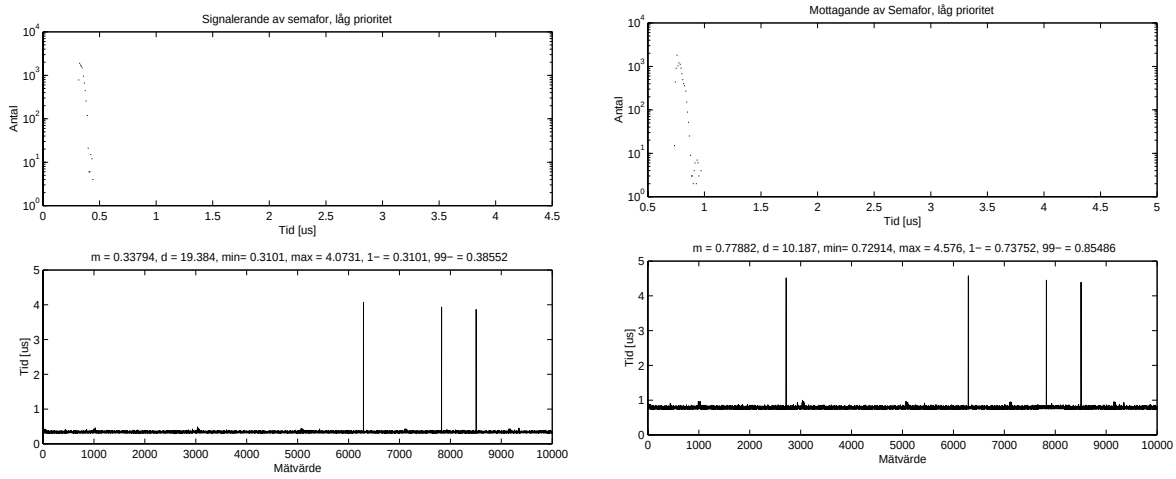
IPC-Mekanism	Basprioritet	Medeltid för sändning [μs]	Medelid för mottagande [μs]
Händelser	Låg	0.52	60.73
Meddelande	Låg	22.09	103.3
Händelser	Hög	0.32	0.76
Meddelande	Hög	0.79	0.20
Händelser	Realtid	0.34	0.08
Meddelande	Realtid	0.78	0.20

Tabell 7 - Jämförelse av tidsåtgång vid meddelandehantering med händelser och meddelanden, belastat system.

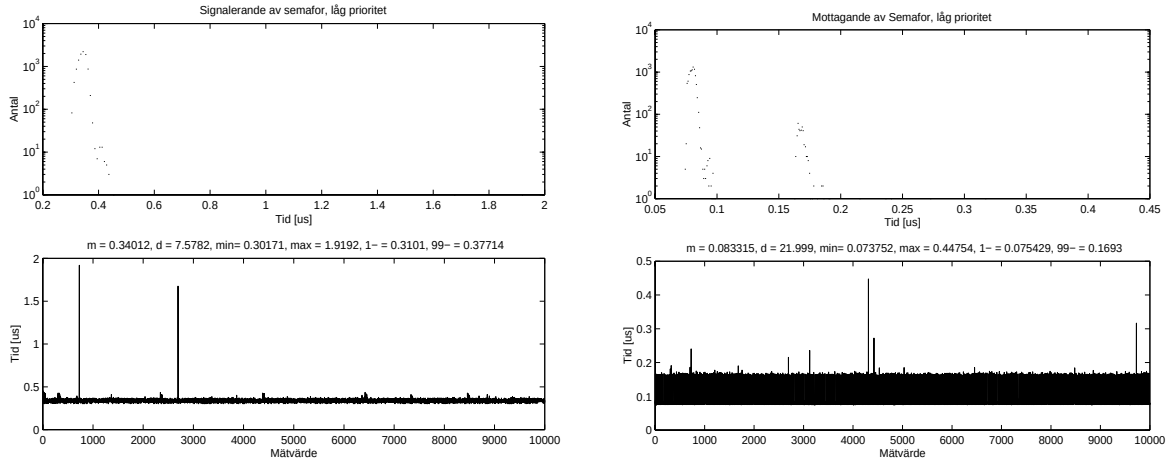
9.3.1.5 Tid för att släppa en semafor, en väntande tråd blir ready

Ännu en metod för interprocess kommunikation är semaforer. De resultat som beskrivs här är resultat av tester beskrivna i 8.5.2, 8.5.3 och 8.5.4 dessa försök är baserade på de som beskrivs i 8.3 och 8.4. Skillnaden mot de tidigare försöken är att dessa försök använder semaforer istället för händelser eller meddelanden.

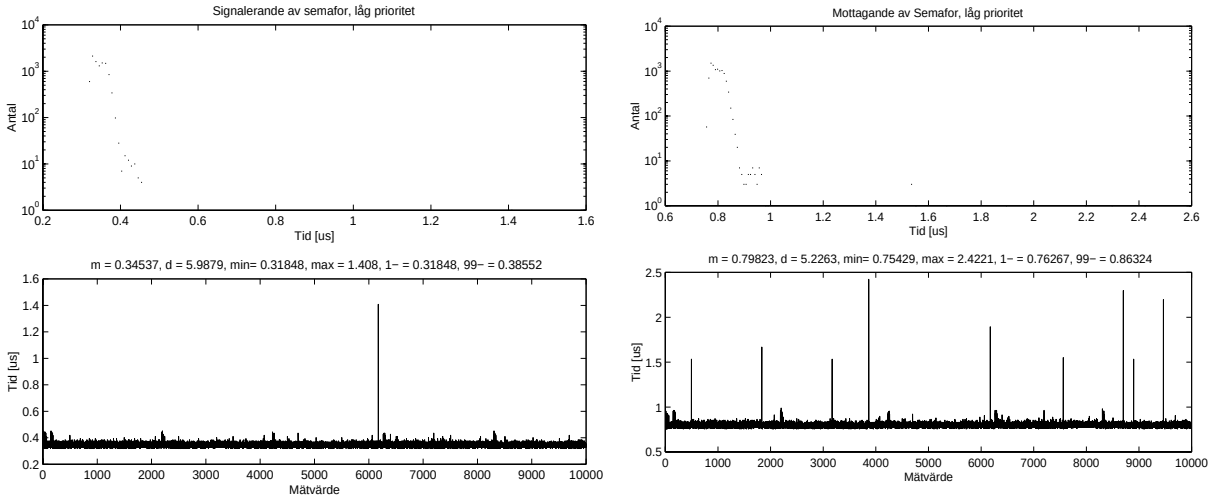
.



Figur 58 - Normal basklass obelastat system, den mottagande tråden har lägre prioritet än den sändande



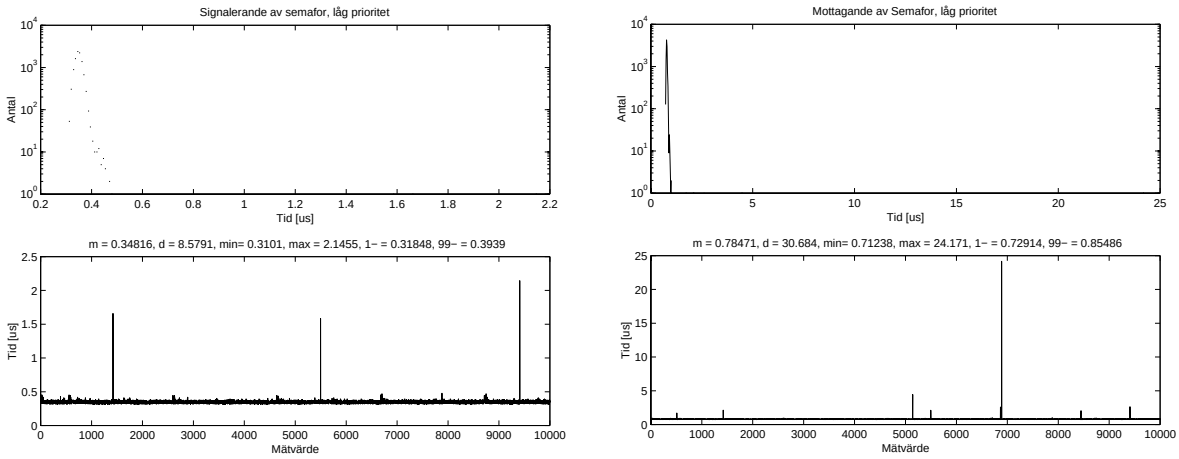
Figur 59 - Hög basklass obelastat system, den mottagande tråden har lägre prioritet än den sändande



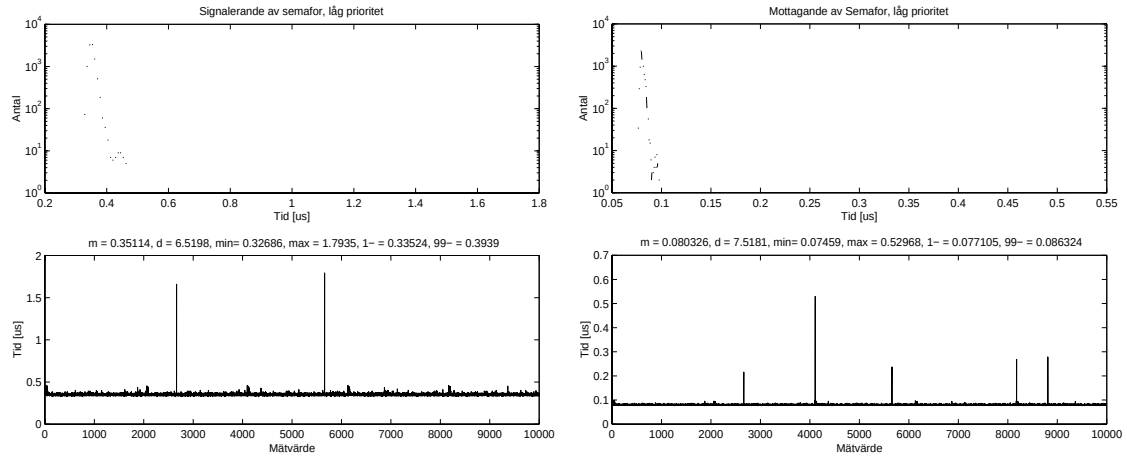
Figur 60 - Realtids basklass obelastat system, den mottagande tråden har lägre prioritet än den sändande

Kommentarer: figur 58, 59 och 60

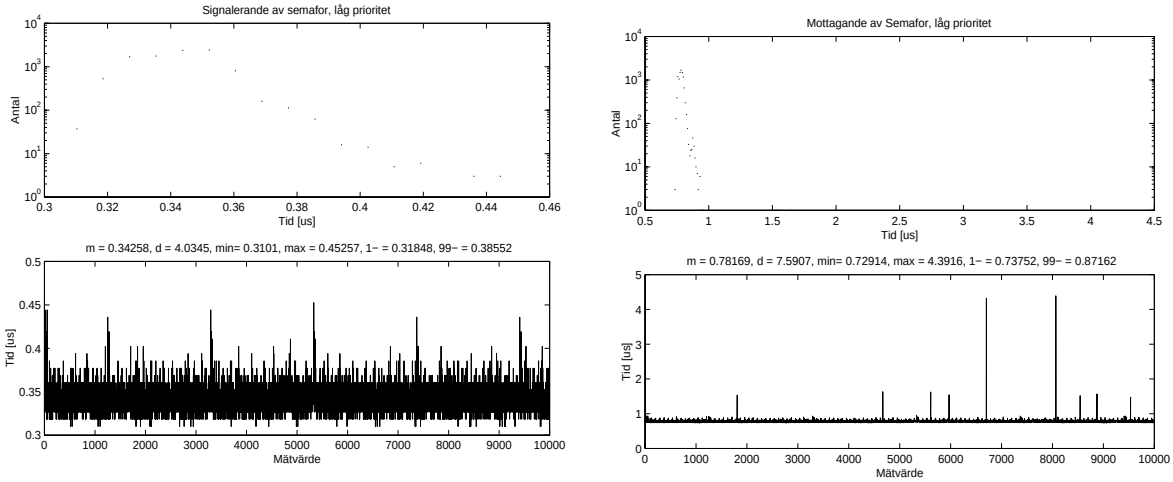
Enligt dessa försök har processens basprioritet ingen inverkan på den tid det tar att signalera en semafor. Tiden för mottagande av en semafor är lika i det normala fallet och i realtidsfallet. Vid hög basprioritet är däremot tiden för mottagande en tiondel av vad den är i de andra fallen vilket är i högsta grad anmärkningsvärt.



Figur 61 - Normal basklass belastat system, den mottagande tråden har lägre prioritet än den sändande



Figur 62 - Hög basklass belastat system, den mottagande tråden har lägre prioritet än den sändande



Figur 63 - Realtid basklass belastat system, den mottagande tråden har lägre prioritet än den sändande

Kommentarer: figur 61, 62 och 63

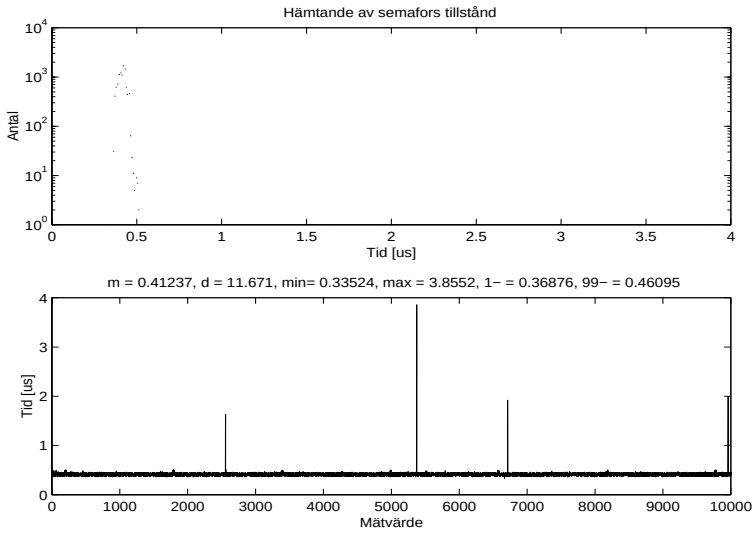
När systemet belastas uppför sig systemet på i stort sätt samma sätt. Tiderna för sändande och mottagande har samma storleksordning. Det lustiga uppträdandet vid mottagande och hög basprioritet kvarstår även i det belastade fallet är tiden för mottagande en tiondel av tiden i de övriga fallen.

De övriga försöken gav lika resultat och presenteras därför i form av figurer i avsnitt 12.1.3: "Semaforhantering"

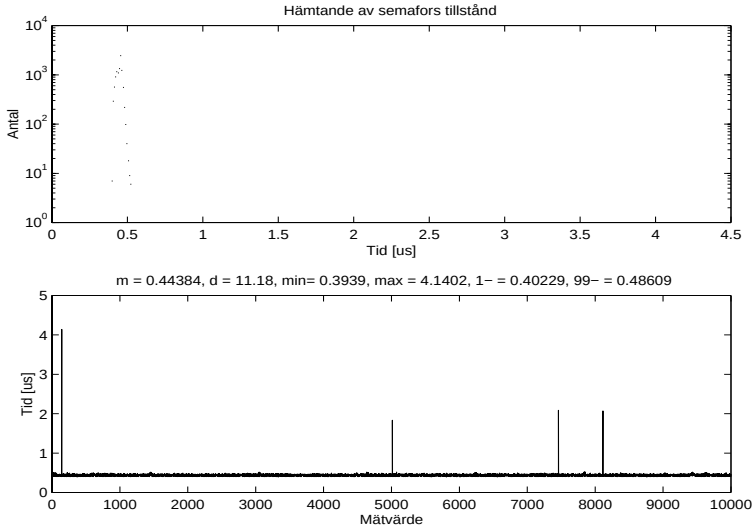
Försöken med semaforer är i högsta grad intressanta, till skillnad från de tidigare fallen där systemet uppträdande förändrats avsevärt vid belastning (se t ex figur 46 och figur 49). Semaforer är alltså relativt opåverkade av basprioriteter och systembelastning.

9.3.1.6 Tid för att få besked om att en semafor är signalerad

I detta stycke presenteras resultaten av försöket beskrivet i avsnitt 8.5.1: "Tid för att få besked om att en semafor är signalerad"..

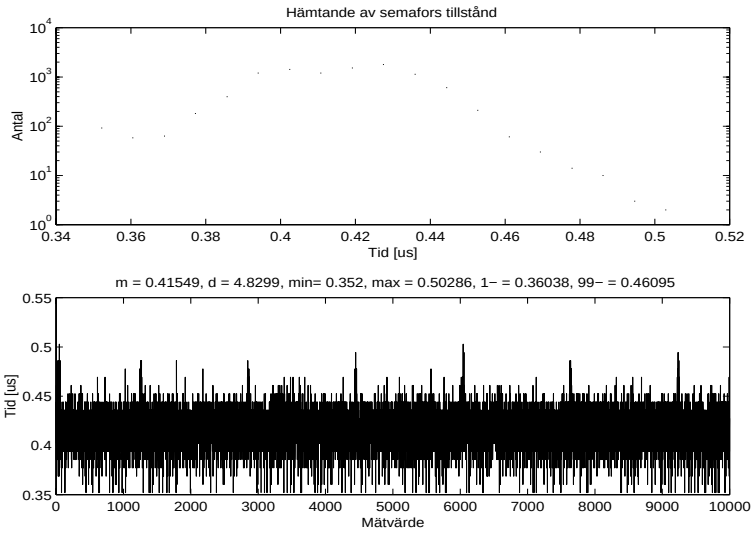


Figur 64 - Normal basprioritet obelastad

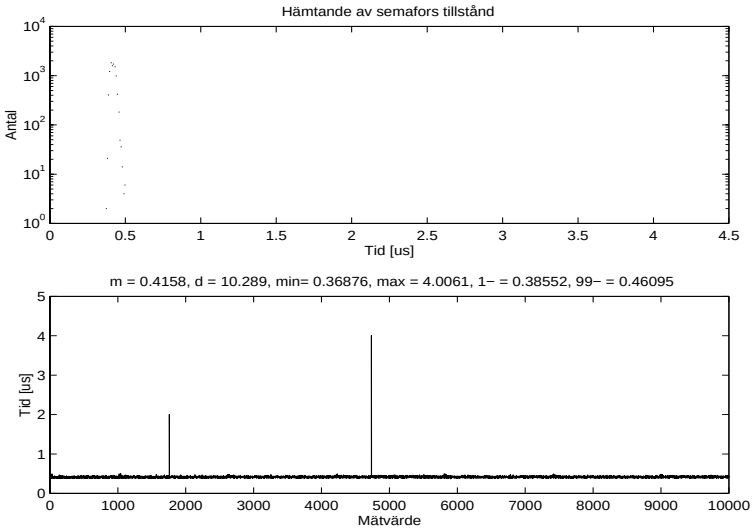


Figur 65 - Normal basprioritet belastad

På normal basprioritet är det ingen större skillnad mellan obelastat och belastat system. I stort sett lika mycket tid går åt i för operationen i bägge fallen.

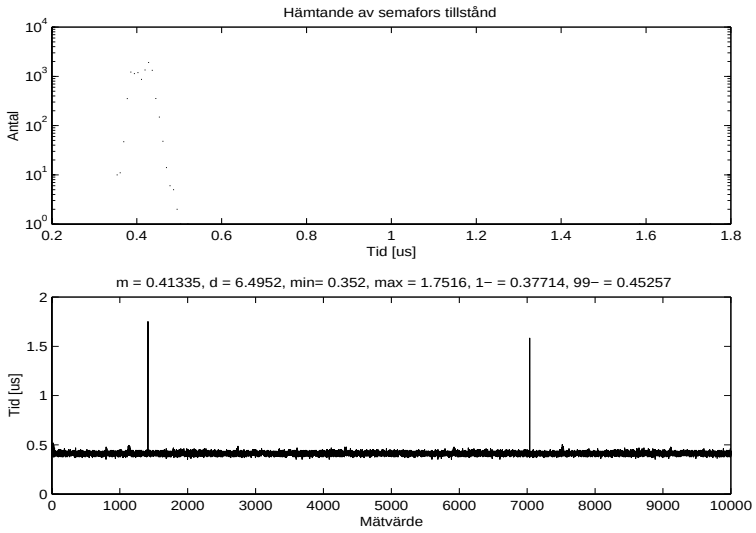


Figur 66 - Hög basprioritet obelastad

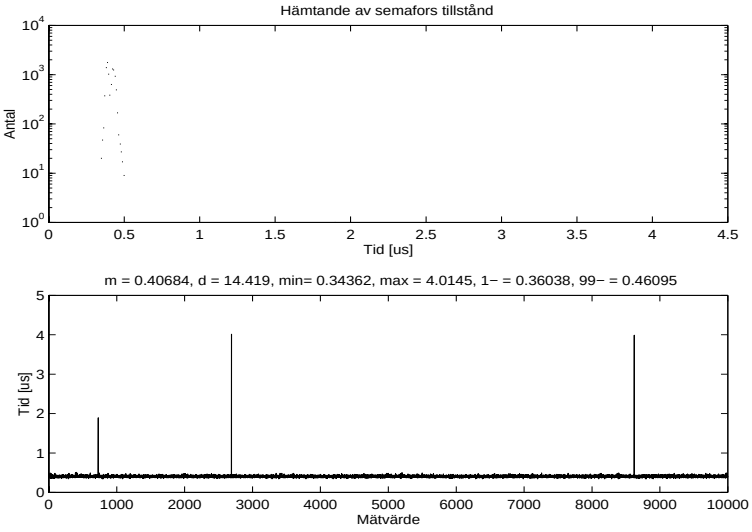


Figur 67 - Hög basprioritet belastad

Inte heller i detta fall är det någon större skillnad mellan testerna. Studerar man diagrammen så ser man att i det belastade fallet förekommer ett par spikar som inte finns med i det obelastade fallet. Om man i stället studerar de beräknade värdena på medel, variationskoefficient samt percentilerna ser man att det inte är någon större skillnad mellan försöken.



Figur 68 - Realtids basprioritet obelastad



Figur 69 - Realtids basprioritet belastad

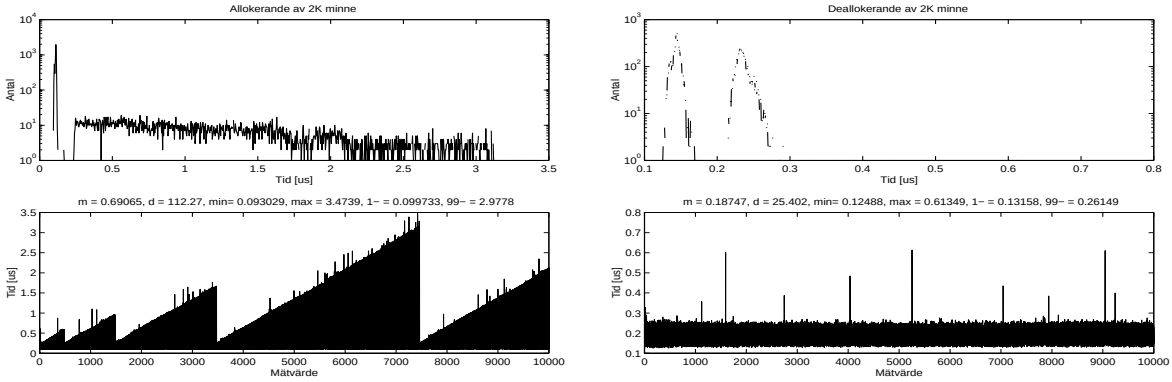
Inte heller i realtids fallet syns någon direkt skillnad mellan de olika fallen.

Jämför man resultaten mellan olika prioritetsnivåer ser man att tiden för att kontrollera en semafors tillstånd i stort sätt är konstant oavsett prioritet och belastning vilket inte var oväntat med tanke på semaforernas uppträdande vid tidigare tester.

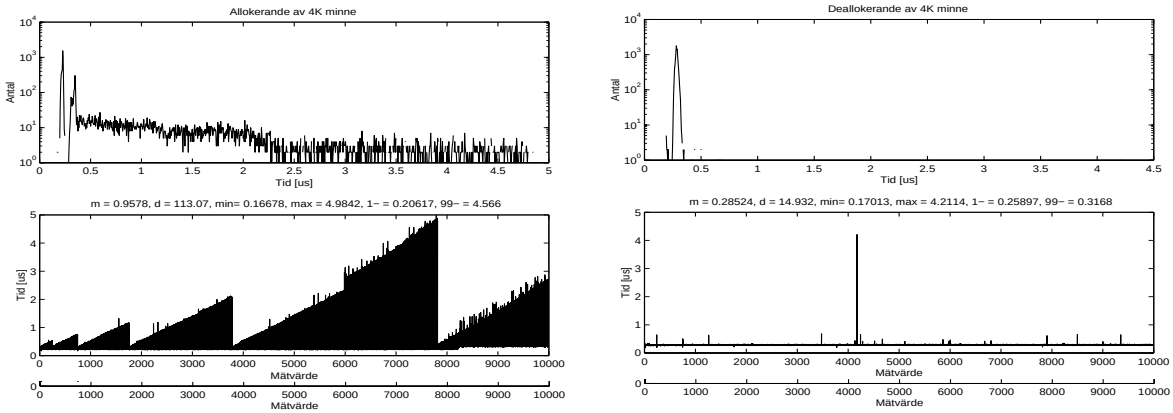
9.3.1.7 Tid för allokering och deallokering av minne

Om man vill vara säker på att tillräckligt minne finns i system så beräknar man en övre gräns på minnesåtgången per process i systemet. Den totala minnesåtgången är sedan summan av processernas minnesåtgång. I ett verkligt system kör man inte alla processer på en gång och man har dessutom inte tillräckligt med minne för att göra det. För att hantera detta använder man dynamiskt minne dvs minne som allokeras och avallokeras under tiden som systemet körs.

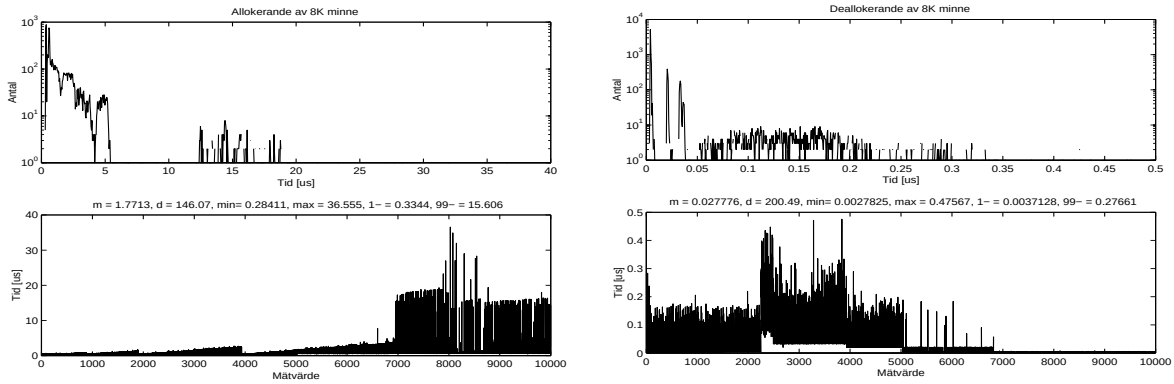
I detta stycke beskrivs resultaten av de tester som beskrivs i avsnitt 8.6: "Minnesallokering".



Figur 70 - Normal basprioritet obelastat system. Allokering av 2K minne.



Figur 71 - Normal basprioritet obelastat system. Allokering av 4K minne.



Figur 72 - Normal basprioritet obelastat system. Allokering av 8K minne.

Figur 70 till 72 säger väldigt mycket om hur NT hanterar processers minne!

Till varje process hör ett block av minne, den så kallade heapen. Det är minne i detta block processen använder då den tex behöver minne temporärt vid kopieringar, det är här många variabler lever och det är från heapen det minne som allokeras dynamiskt tas.

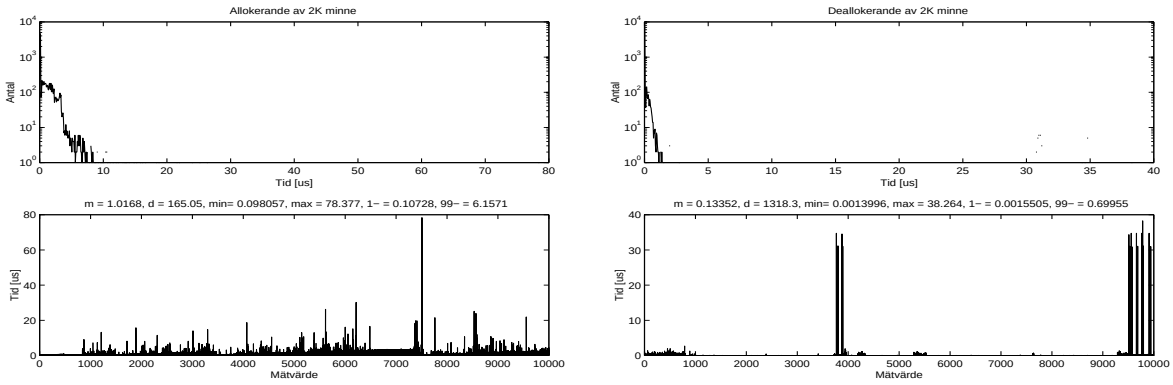
I ett operativsystem är det relativt dyrt att reallokera minne dvs göra det allokerade området lite större. Systemet måste först hitta ett nytt ledigt minnesområde som kan användas. Informationen ska sedan kopieras från det gamla minnesområdet till det nya. För att undvika detta brukar man allokera lite mer minne än vad man tror ska gå åt så när en begäran om reallokering sker har man redan minne tillgängligt och reallokeringen kan ske relativt enkelt.

Frågan är då hur mycket heap ska man initialt allokera till varje process? Standardvärdet i NT är 1MB. En annan intressant fråga är vad man gör när heapen tagit slut och måste göras större. Det finns en tumregel som säger att när man allokerar om minne så ska man göra det nya minnesutrymmet minst dubbelt så stort som det gamla. Man kan då hantera fallet med en kopiering av hela det ursprungliga minnesutrymmet (vilket är det “värsta” som kan hända) utan att behöva allokera om minne ännu en gång.

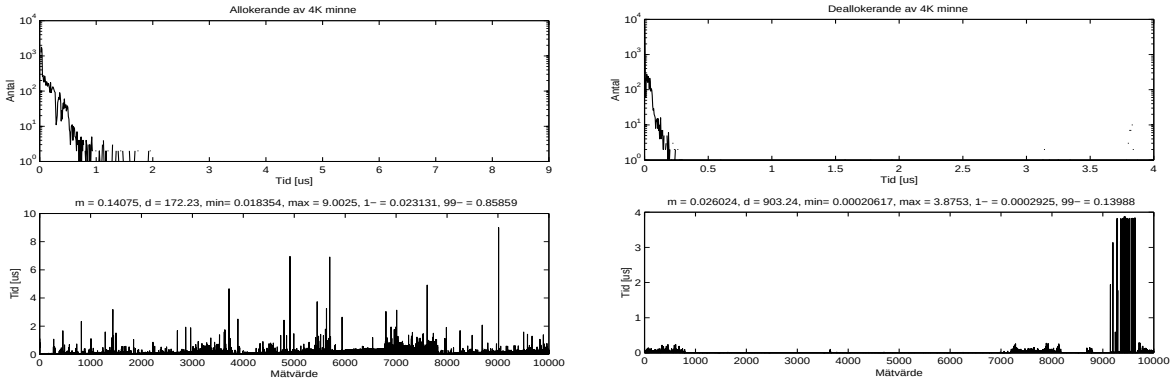
Studerar man grafen så ser man att tiden för allokering av minnet växer linjärt till en viss punkt då den plötsligt går tillbaka till någon form av lägsta nivå. För att sedan börja växa linjärt igen. Studerar man längden på “trappstegen” efter tidsaxeln så ser man att längden på stegen i stort sätt fördubblats

Testprogrammet allokerar 1byte minne i taget och upprepar detta många gånger. Av graferna att döma söker minneshanteraren linjärt i heapen efter nästa lediga position att allokera. När det är slut på utrymme i heapen allokeras dubbelt så mycket och minnesallokeringen fortsätter i det nya området i heapen.

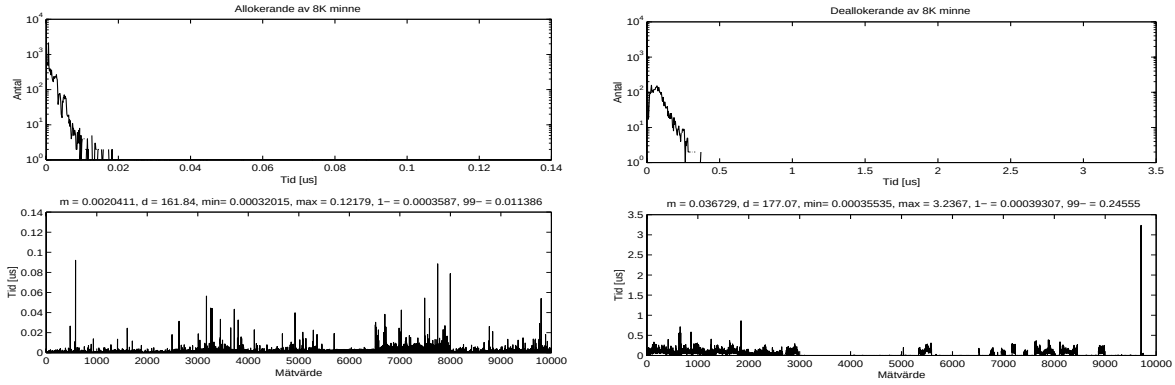
Tiden för deallokering av minne är i stort sett konstant. På samma sätt som i övriga tester förekommer enstaka spikar på grund av att systemprocesser exekveras.



Figur 73 - Normal basprioritet belastat system

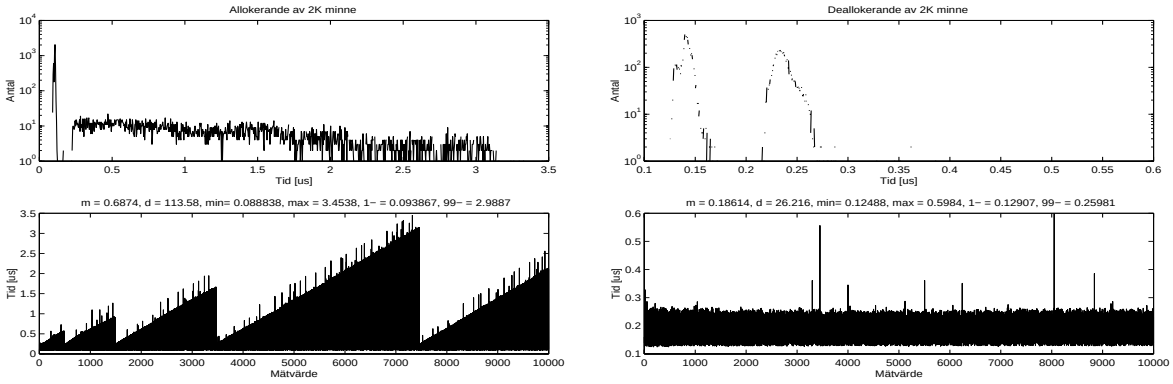


Figur 74 - Normal basprioritet belastat system

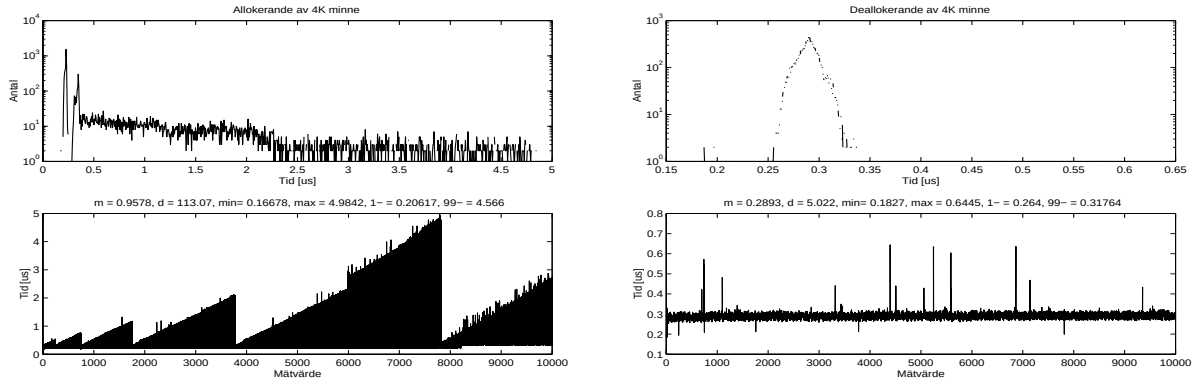


Figur 75 - Normal basprioritet belastat system

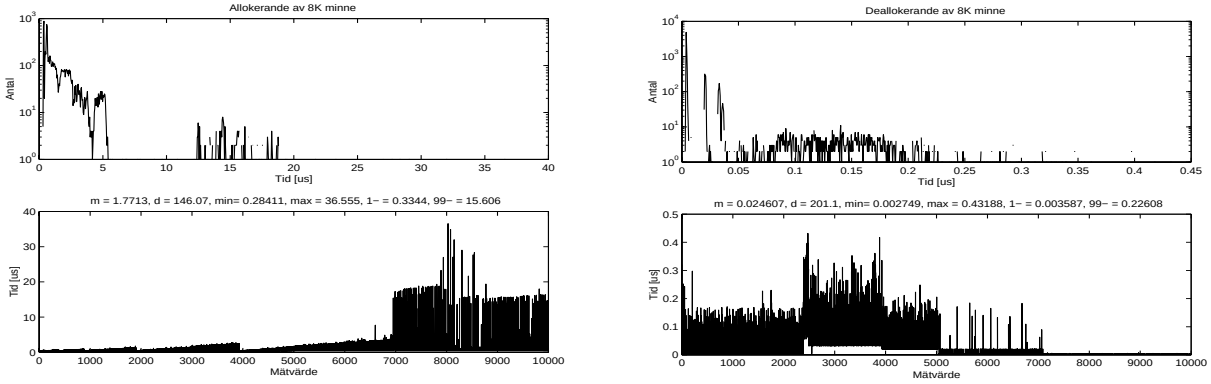
När systemet belastas av en process på samma prioritet som testprogrammet ser vi samma mönster som tidigare.



Figur 76 - Hög basprioritet obelastat system

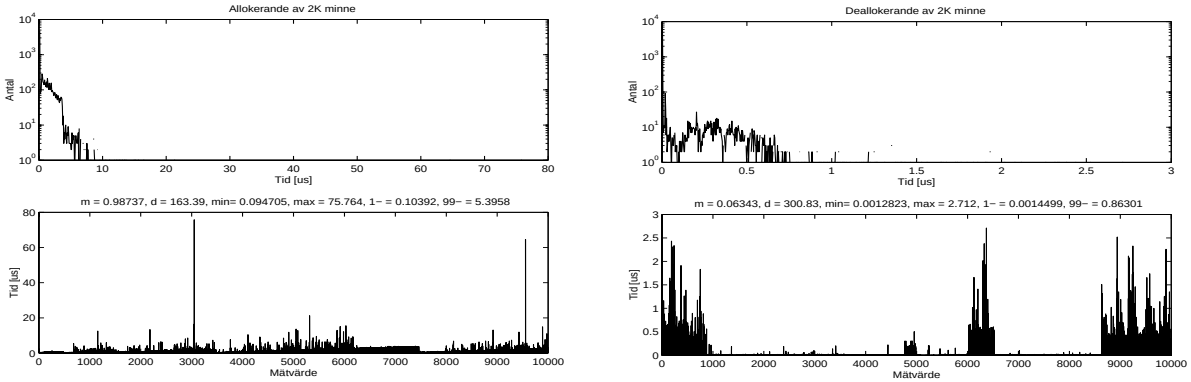


Figur 77 - Hög basprioritet obelastat system

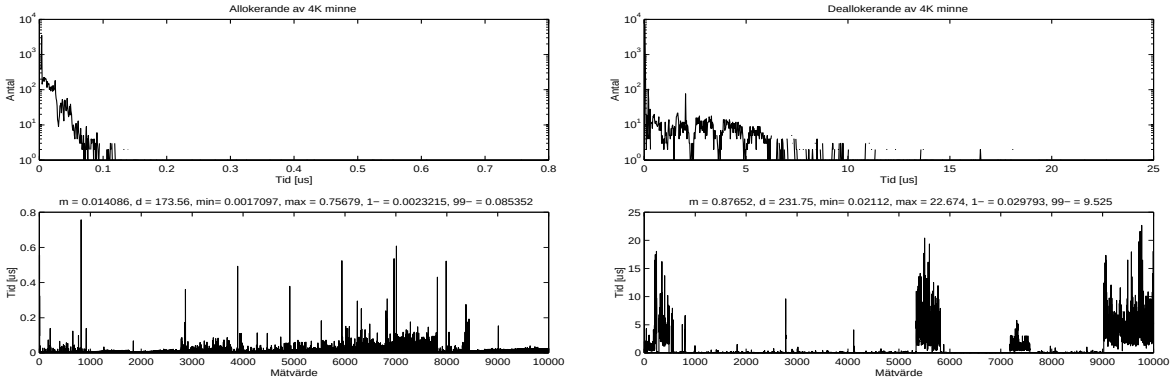


Figur 78 - Hög basprioritet obelastat system

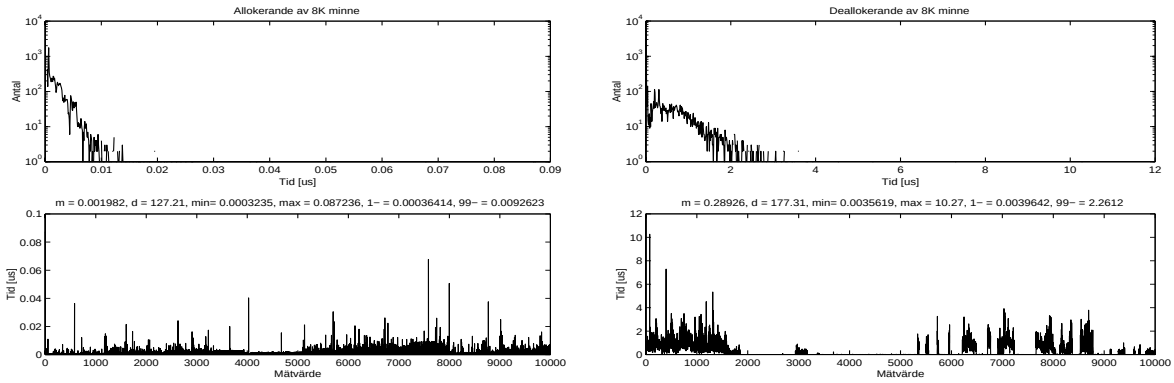
Resultaten här är principiellt samma som i fallet med normal basprioritet. Värt att notera är att i slutet av 8K testet ökar svarstiden med en faktor 10. Detta sker både i fallet med normal och hög basprioritet.



Figur 79 - Hög basprioritet belastat system

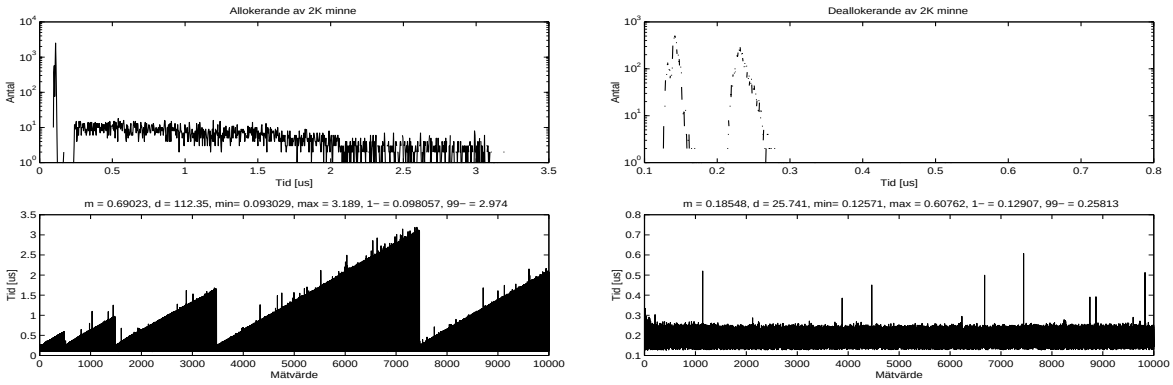


Figur 80 - Hög basprioritet belastat system

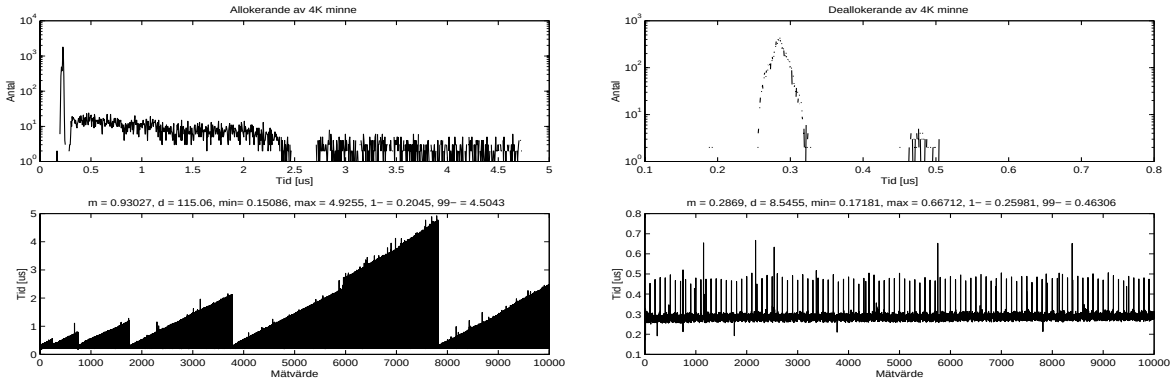


Figur 81 - Hög basprioritet belastat system

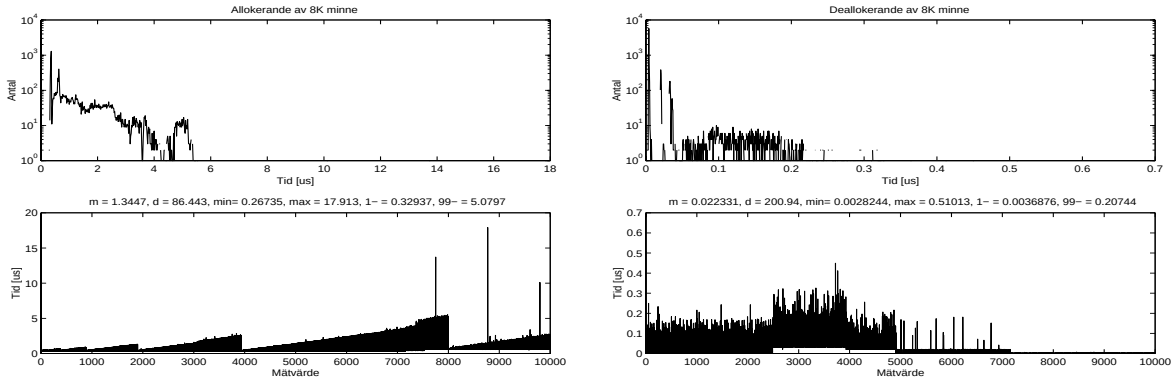
På hög basprioritet är inte inverkan av belastningen lika stor som vid normal prioritet men en hel del spikar förekommer fortfarande.



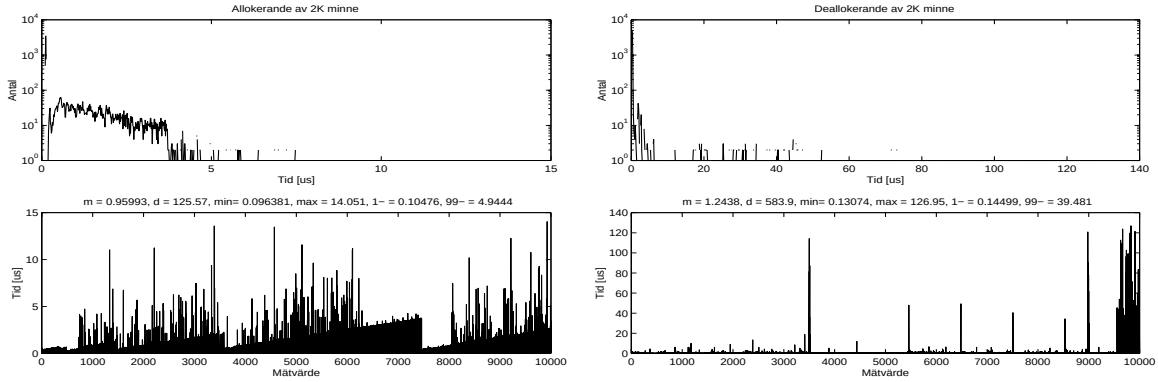
Figur 82 - Realtids basprioritet, obelastat system



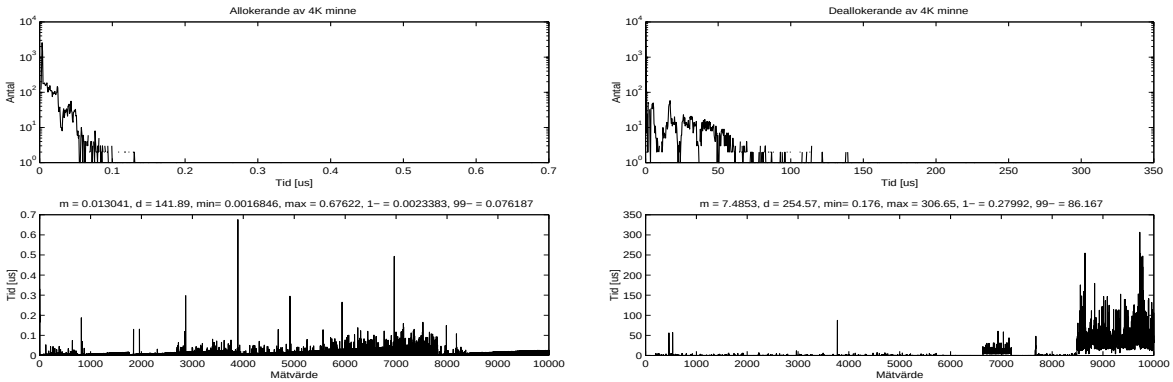
Figur 83 - Realtids basprioritet, obelastat system



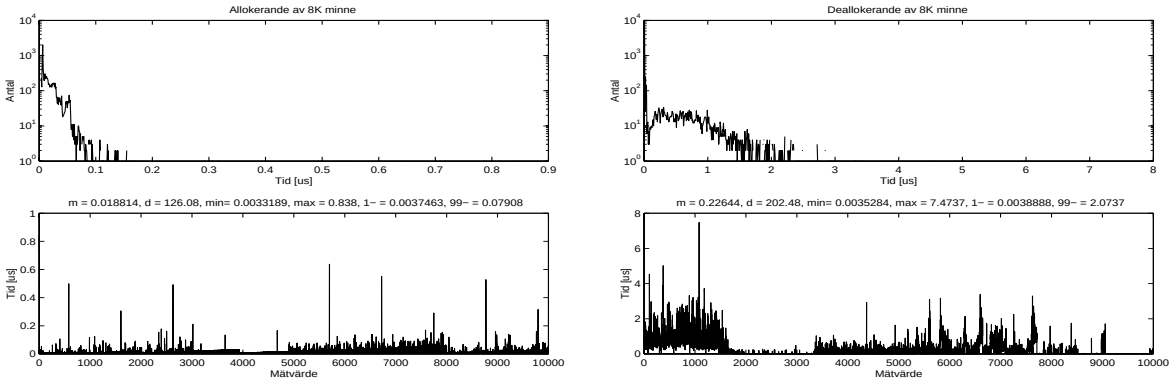
Figur 84 - Realtids basprioritet, obelastat system



Figur 85 - Realtids basprioritet belastat system



Figur 86 - Realtids basprioritet belastat system



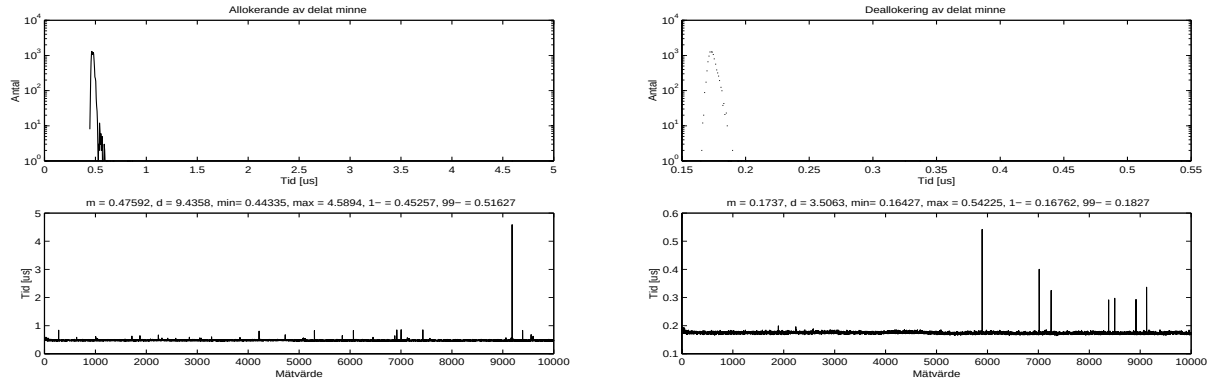
Figur 87 - Realtids basprioritet belastat system

Realtidsfallet ser i stort sett likadant ut som de övriga fallen.

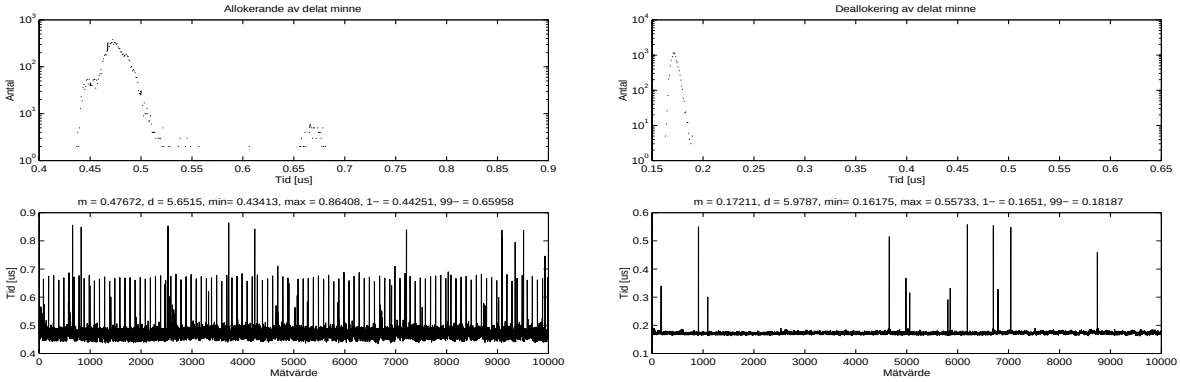
Om man jämför de obelastade testerna på de olika prioritetsnivåerna så ser man att de är principiellt identiska. Det tar tex ca 0.95 µs att allokera 4kb minne oavsett prioritet. En möjlig förklaring till detta kan vara att minnesallokering inte är prioritetsberoende. Operativsystemet hanterar minnesalokering på samma sätt oavsett vilken tråd som efterfrågar minne och oberoende av trådens prioritet. En tråd med hög prioritet får inte minne allokerat snabbare än en tråd med lägre prioritet. Däremot spelar prioriteten en stor roll vid själva schemaläggningen av processerna. En tråd med lägre prioritet kan bli avbruten vid minnesallokering vilket är märkbart i de belastade fallen.

9.3.1.8 Allokering av delat minne

I detta stycke beskrivs resultaten av det test som beskrivs i avsnitt 8.6.2: "Allokering av delat minne".

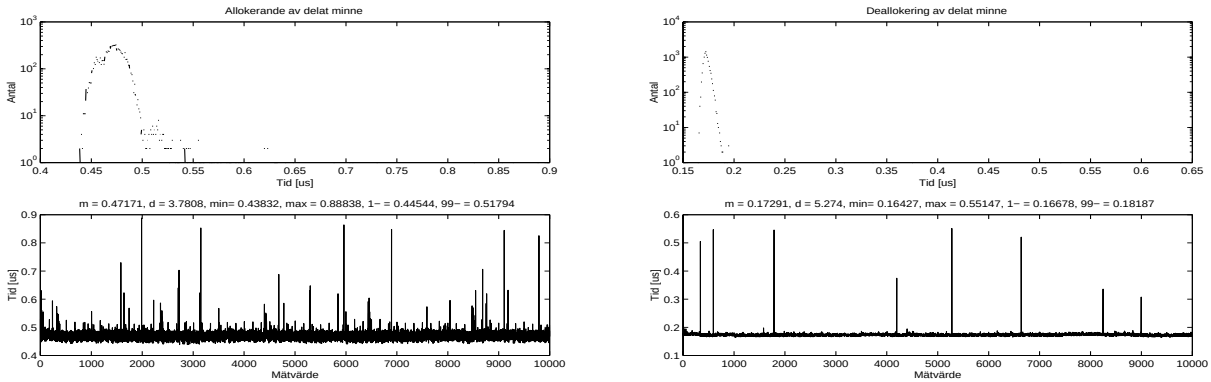


Figur 88 - Normal basprioritet, obelastat system

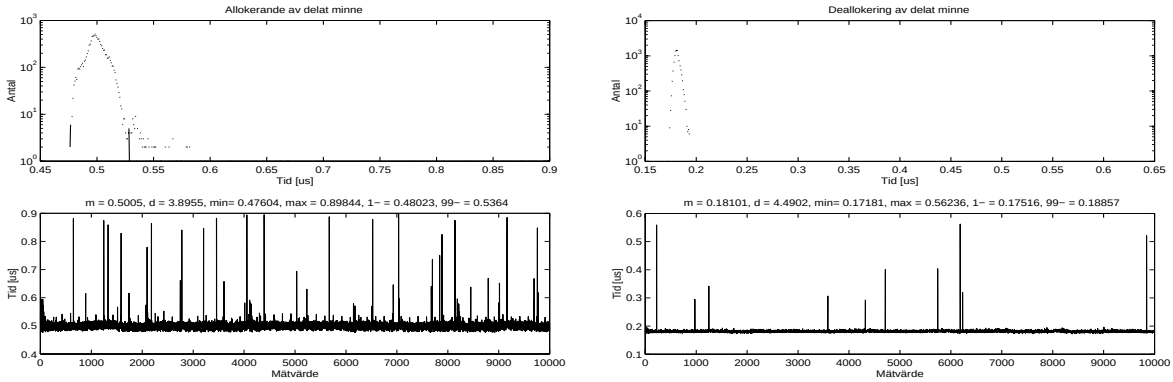


Figur 89 - Normal basprioritet, belastat system

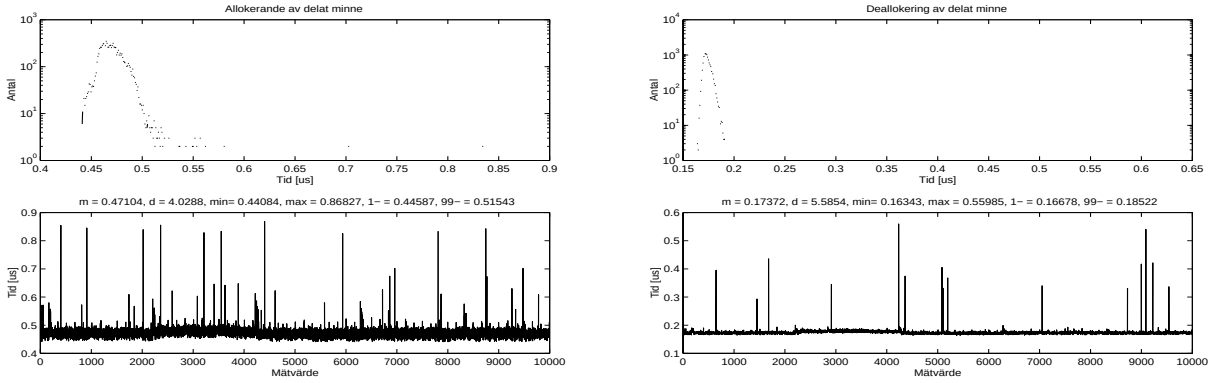
Förutom ett större antal spikar så skiljer sig inte fallet med belastat system sig nämnvärt från det obelastade. I bägge fallen tar det ca 0.476 µs att allokera minnet och 0.173 µs att deallokera minnet.



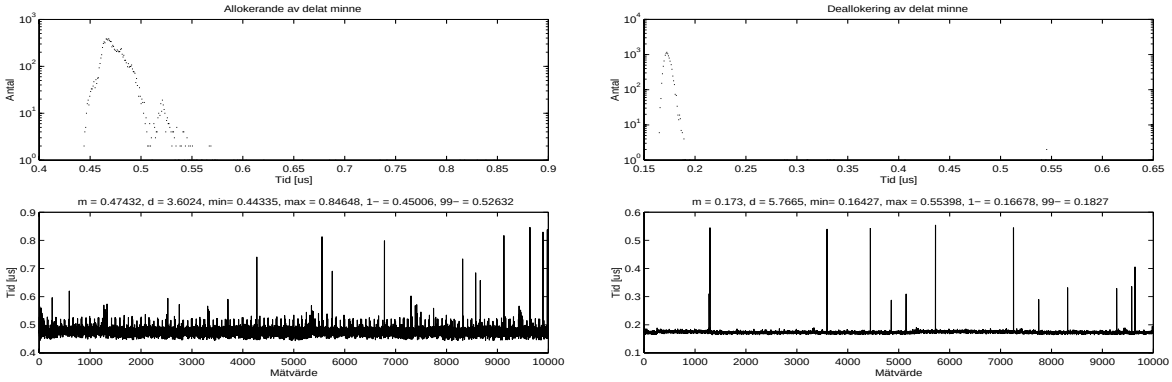
Figur 90 - Hög basprioritet, obelastat system



Figur 91 - Hög basprioritet, belastat system



Figur 92 - Realtids basprioritet, obelastat system



Figur 93 - Realtids basprioritet, belastat system

Inte heller på hög och realtids basprioriteter är det någon större skillnad mellan de olika fallen.

Man kan dra slutsatsen att inte heller allokering av delat minne påverkas nämnvärt av den allokerande processens prioritet. Systemets belastning har en viss inverkan men den yttrar sig främst genom ett mindre antal spikar.

Analys

10 Resultat och slutsatser

I detta kapitel redogörs för i vilken grad de uppsatta målen uppnåtts samt vilka slutsatser man kan dra av arbetet.

10.1 Resultat

Har de resultat som beskrevs i syftet uppnåtts?

10.1.1 Beskrivning av RTOS

Realtidsoperativsystemen pSOS från Integrated Systems Inc. och OSE från Enea har översiktligt beskrivits.

10.1.2 Identifiering av mekanismer i RTOS

Några mekanismer som kan anses vara extra viktiga vid konstruktion av realtidssystem har identifierats.

10.1.3 Redovisning av NTs uppbyggnad

En beskrivning av Windows NT har gjorts. Speciell tyngdpunkt ligger på schemaläggaren då förståelse av denna är central vid användandet av NT i sammanhang då man har realtids eller prestanda krav.

10.1.4 Genomförande av praktiska tester

Tester har utförts av de realtidsmekanismer som tidigare identifierats. Dessa tester har utförts på olika prioritetsnivåer och med olika systembelastning. Resultaten av testerna har sedan sammanställts och analyserats.

10.2 Slutsatser

Vilka slutsatser kan man då dra utifrån detta arbete?

För få statiska prioritetsnivåer

Antalet statiska prioritetsnivåer i Windows NT är för få för att det ska vara praktiskt möjligt att använda algoritmer som tex RMS. NTs 16 nivåer kan jämföras med de 256 prioritetsnivåer som pSOS erbjuder.

Begränsad upplösning på timers

Centralt i realtidssammanhang är tillgången av tillförlitliga timers, då många processer ska köras periodiskt. Den bästa timermekanismen i NT är de sk *multimedia-timers* men även de har en begränsad upplösning på 1 millisekund. Som man kan se i *avsnitt 9.3.1: "Normal timer"* och *avsnitt 9.3.1.1: "Multimedia-timers"* så är inte precisionen den bästa. När systemet belastas existerar tillfällen då den verkliga fördröjningen är 3 till 4 gånger längre (minst) än avsett. Detta medför att det är olämpligt att använda timers för att tex polla hårdvara med jämna mellanrum. Hårdvaran är tvungen att antingen vara rejält buffrad så att det inte gör något om man blir lite sen ibland eller också måste hårdvaran använda avbrott.

Multimediatimers beteende är HAL beroende

Om man ska använda multimediatimers och har krav på små periodtider finns ett starkt beroende mellan dessas uppträdande och den HAL som systemet exekverar på. Som man kan se i avsnitt 7.4: "Multimedia timers" så ger systemet den avsedda fördröjningen i snitt men de faktiska fördröjningarna avviker mycket från de avsedda. Detta medför att NTs multimediatimers är oanvändbara för t.ex. sampling redan vid frekvenser runt 1000 Hz. Dessa timers beror dessutom på viket HAL som är installerat detta medför att ett system som fungerar på en hårdvara kan sluta fungera t ex vid byte till en multiprocessorarkitektur.

Systemanropen ej begränsade i tidsåtgång

I manualerna för NT finns inget som anger någon maximal tid som ett anrop kan ta. För vissa anrop kan man i och för sig sätta en timeout men det är långt ifrån alla anrop som stödjer detta.

Studerar man hur lång tid olika anrop och operationer tar (se avsnitt 9.3: "Testresultat"), så ser man att tidsåtgången är väldigt varierande. Det finns alltså inget sätt att på förhand säga hur lång tid ett anrop tar. Detta medför att det är omöjligt att på förhand försäkra sig om att uppsatta tidskrav möts.

Ickedeterministiska rutiner för hantering av priority inversion

Ett realtidssystem måste ha mekanismer för att hantera priority-inversion. Dessa mekanismer måste dessutom ha ett deterministiskt beteende. Den algoritm som valts i NT går ut på att den tråd som inte exekverat på ett tag får en slumpmässig höjning av sin prioritet. Detta fungerar utmärkt i ett vanligt operativsystem men det är inte tillräckligt bra för ett realtidssystem.

Val av drivrutiner

Då avbrott hanteras av drivrutiner skrivna av hårdvarans tillverkare är den tid som de är bortmaskade inte bestämd på förhand utan upp till de olika drivrutinsförfattarna. Ett bra exempel på detta är drivrutinen för diskar på IDE-bussen som i vissa fall maskar bort andra avbrott i upp till 5ms.

Denna hårda koppling mellan tredjeparts-drivrutiner och systemets uppförande medför att det är nödvändigt att noggrant inventera vilka drivrutiner man har i sitt system och vad de betyder för systemets uppträdande. Denna granskning måste genomföras varje gång någon drivrutin uppdateras.

En direkt rekommendation är också att undvika hårddiskar som ansluts via IDE-bussen om man har ett system där realtidsprestanda är viktigt.

Vanliga timers fungerar tillfredsställande vid låg belastning och periodtid > 100 ms

Vid en periodtid på 100 ms är det relativa felet hos den vanliga timern relativt liten i det obelastade fallet. När systemet belastas är man tvungen att öka processens basprioritet för att bibehålla denna noggranhet. Den vanliga timermekanismen kan med fördel användas när den exakta periodtiden inte är extremt viktig.

Multimediatimers är bättre än vanliga och rel. okänsliga för belastning

Har man krav på lägre periodtider och/eller inte möjlighet att höja basprioriteten på sina processer är multimediatimers ett gott alternativ. Man ska dock vara medveten om att den algoritm som används kan ge effekten att periodtiden enbart stämmer i medeltal. Har man maximal otur får man två händelser åt gången och dubbel periodtid mellan grupperna med händelser.

Ett exempel där medelperiodtiden ser bra ut men mätvärdena i princip har en tvåpunktsfördelning på var sin sida medelvärdet kan t.ex. ses i figur 29.

Lämplig prioritetsnivå viktigt vid belastat system

Genomgående för många tester var att processernas basprioriteter hade stor inverkan på systemets uppförande. Hur olika processer ska fördelas mellan olika basprioritetsnivåer är något som måste ingå i systemdesignen.

Genom ett genomtänkt användande av basprioritetsnivåerna har man stora möjligheter att konstruera ett system med goda prestanda och ett relativt deterministiskt beteende.

Semaforer är okänsliga för basprioritet och belastning

Ett av undantagen från den kraftiga inverkan av systembelastning och basprioriteter var semaforerna. I försöken uppvisade semaforerna ett beteende som var i stort sätt opåverkat av desa faktorer. Semaforer är alltså att föredra framför händelser för att synkronisera olika processer.

Undvika att allokera dynamiskt minne i den tidskritiska delen av programmet

Tidsåtgången vid minnesallokering är inte konstant utan starkt kopplad till mängden minne som alokerats. Man bör undvika att allokera minne dynamiskt i tidskritiska delar av systemet utan allokera detta i förväg.

NT lämpligt för mjuka RT-system

En konsekvens av de tidigare slutsatserna är att NT inte är lämplig att använda för hårda realtidssystem då man inte kan garantera att de ställda tidskraven uppfylls.

Däremot är NT som klippt och skuren för vissa mjuka realtidssystem. Till NT finns en rik flora av mät och styrkort, applikationer för databearbetning och lagring. Om dessa kort har interna buffertar och drivrutinerna är välskrivna och inte blockerar andra avbrott onödigt länge kan ett system baserat på NT få goda egenskaper. Man har då flyttat över de delar av som har realtidskrav på sig till hårdvara och därigenom avlastat det övriga systemet.

De absoluta krav tillämpningen ställer på systemet påverkar självklart vilka mekanismer som är användbara. Om de krav man ställer på t ex timers inte är särskilt höga utan det handlar om periodtider på någon sekund och man kan tolerera en avvikelse i storleksordningen 10 ms så fungerar multimedia timers troligen tillfredsställande. En illustration av detta kan ses i figur 45 där multimedia timers med en periodtid på 1s presenteras, här ligger 1-percentilen på 994.5 ms och 99-percentilen på 1005.5 ms jämfört med den ideala tiden 1000 ms.

En lämplig plats för NT är antagligen i periferin av realtidssystemet och för tillämpningar som insamling och analys av mät och testinformation. NT går att använda i applikationer där höga krav på prestanda krävs. Men man har ingen möjlighet att visa att kraven alltid uppfylls. Det enda man kan göra är att minska sannolikheten för detta genom att överdimensionera systemet vad gäller processor, minne och hårddiskprestanda.

11 Referenser

[Krishna & Shin 97] C. M. Krishna & Kang G. Shin, *"Real-Time Systems"*, Mc-GRAW-HILL, ISBN 0-07-114243-6

[Solomon 98] David A. Solomon, *"Inside Windows NT Second Edition"*, Microsoft Press, ISBN1-57231-677-2

[Silberschatz & Galvin 94] Abraham Silberschatz & Peter B. Galvin, *"Operating System Concepts, Fourth Edition"*, Addison-Wesley Publishing Company, ISBN0-201-59292-4

[ISI98] pSOSystem, *"pSOSystem Advanced Concepts"*, pRISM+ / PowerPC CD-ROM Reference Library, part number 090-5030-002

[OSE98] Users Guide R 1.0.0 OSE Concepts

[Jones & Regehr 98] Michaek B. Jones & John Regehr, *"Issues In Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT"*, Proceedings of NOSSDAV 98

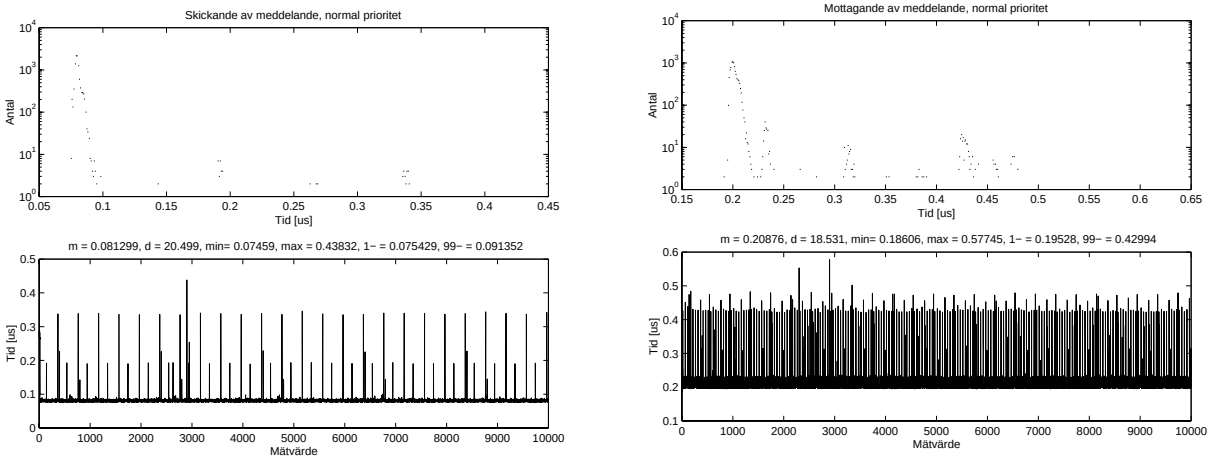
Referenser

12 Bilagor

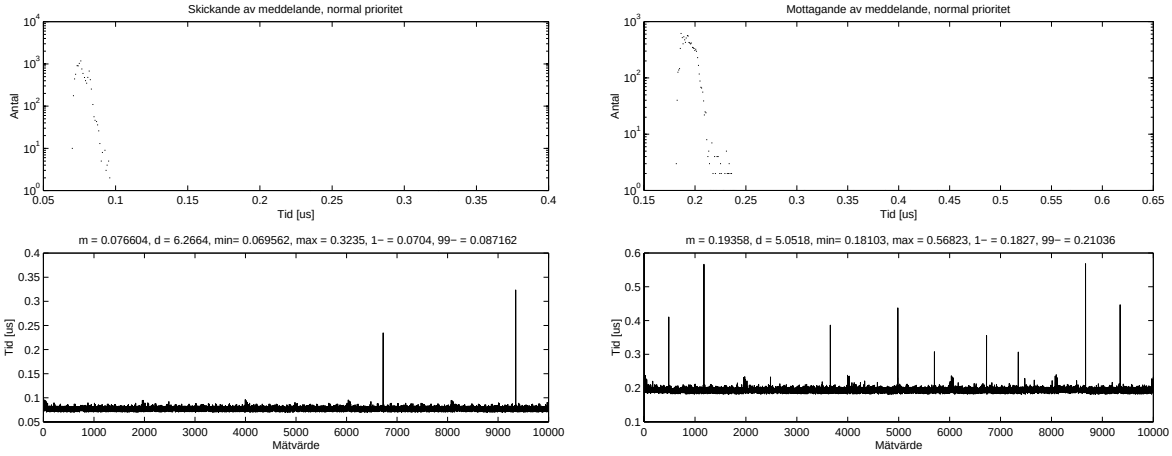
12.1 Figurer

12.1.1 Tid för att skicka ett meddelande

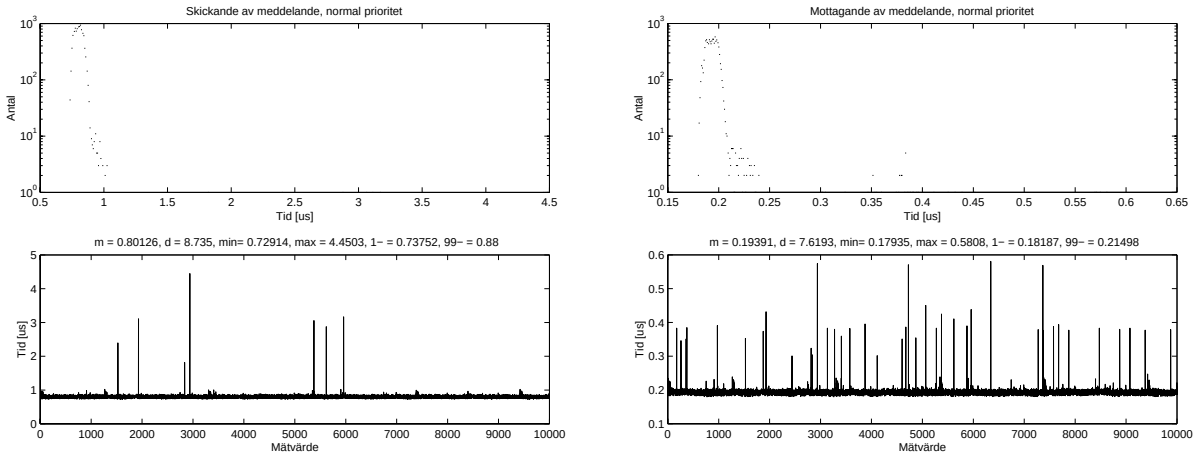
.



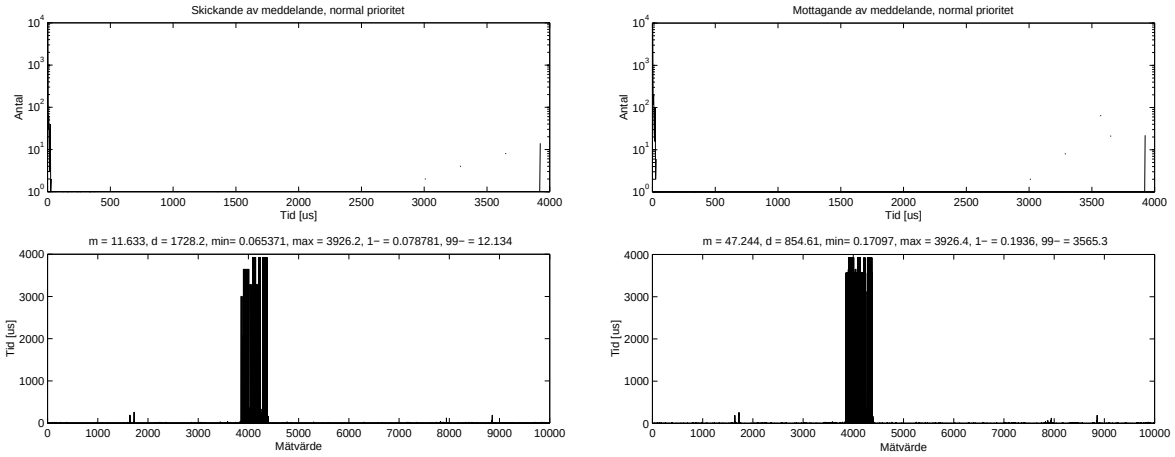
Figur 94 - Normal basprioritet, obelastat system, mottagande tråd har samma prioritet som den sändande



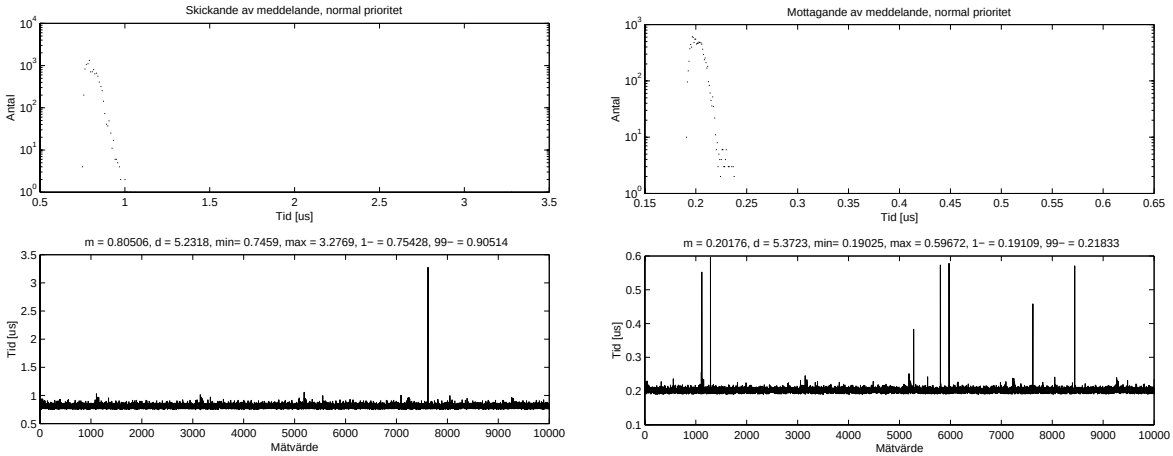
Figur 95 - Hög basprioritet, obelastat system, mottagande tråd har samma prioritet som den sändande



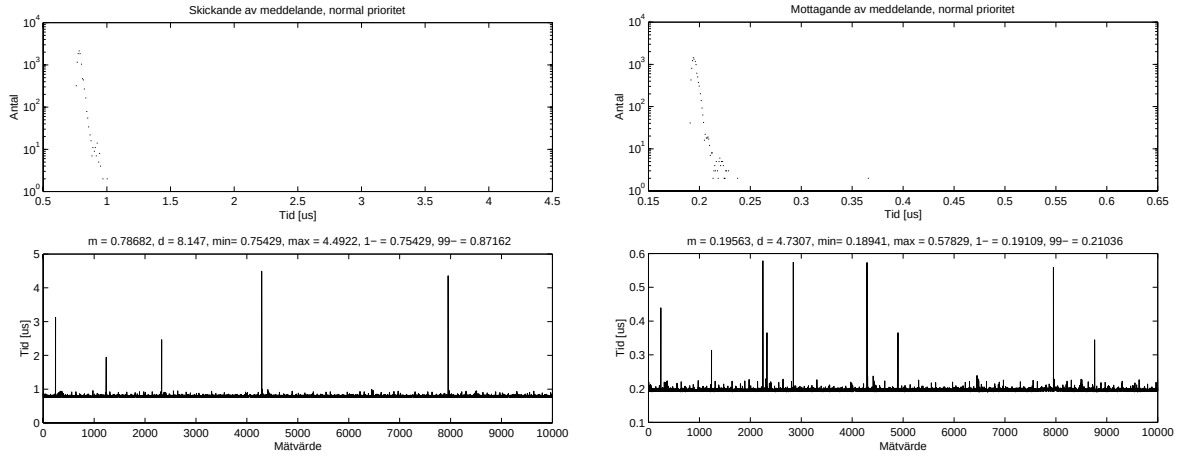
Figur 96 - Realtids basprioritet, obelastat system, mottagande tråd har samma prioritet som den sändande



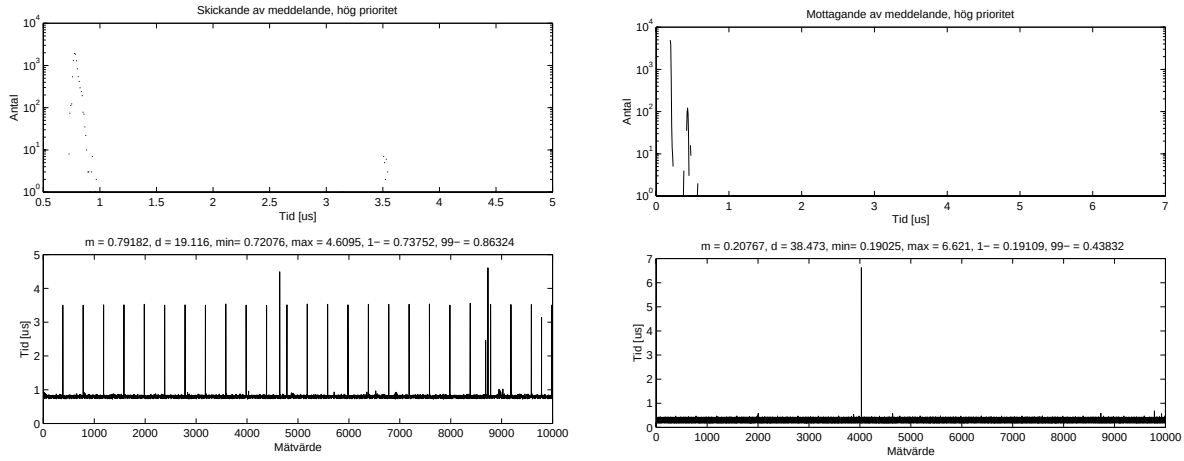
Figur 97 - Normal basprioritet, belastat system, mottagande tråd har samma prioritet som den sändande



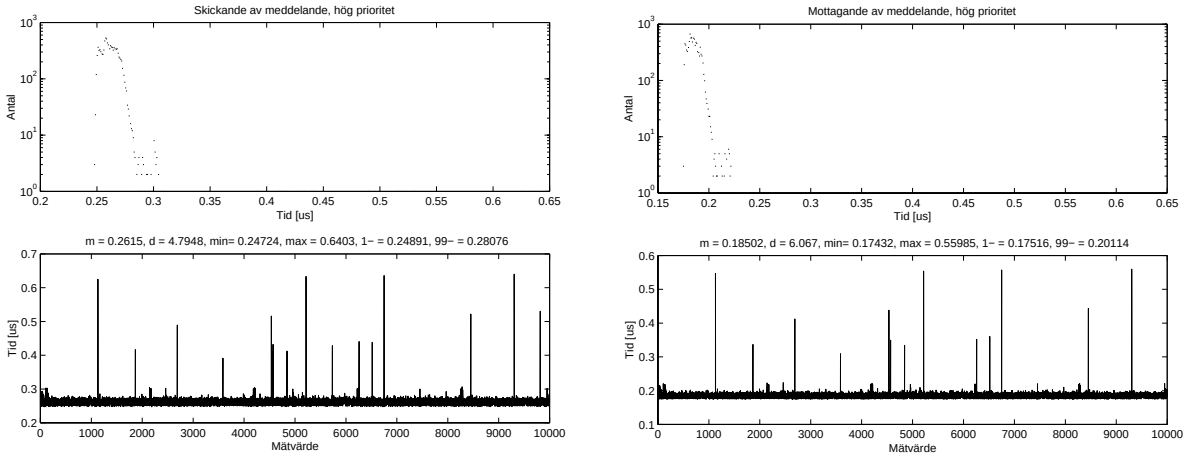
Figur 98 - Hög basprioritet, belastat system, mottagande tråd har samma prioritet som den sändande



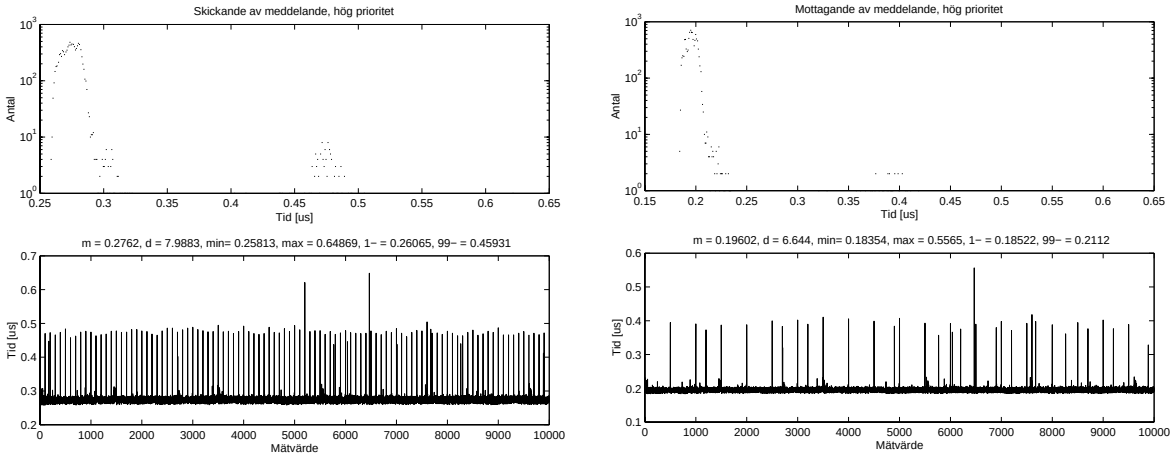
Figur 99 - Realtids basprioritet, belastat system, mottagande tråd har samma prioritet som den sändande



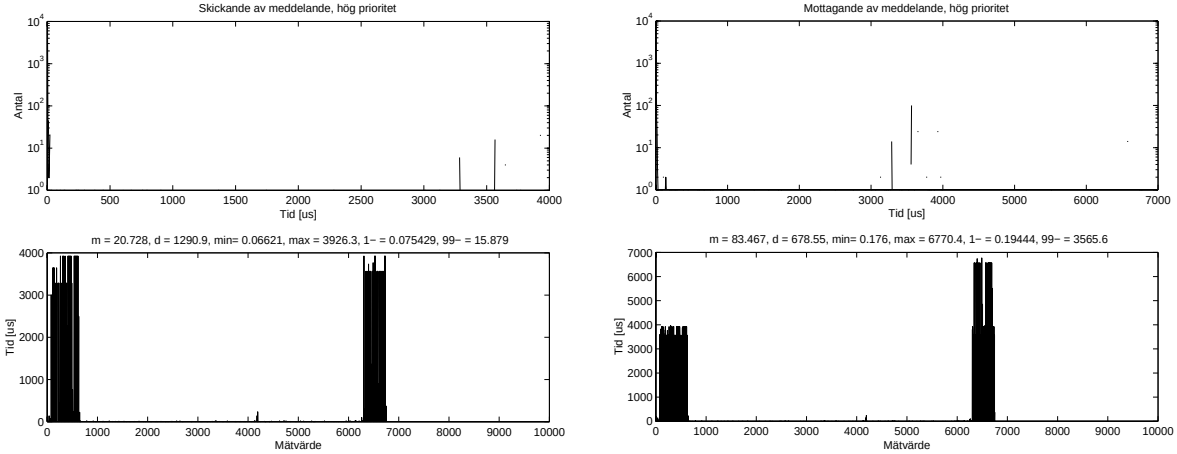
Figur 100 - Normal basprioritet, obelastat system, mottagande tråd har högre prioritet än den sändande



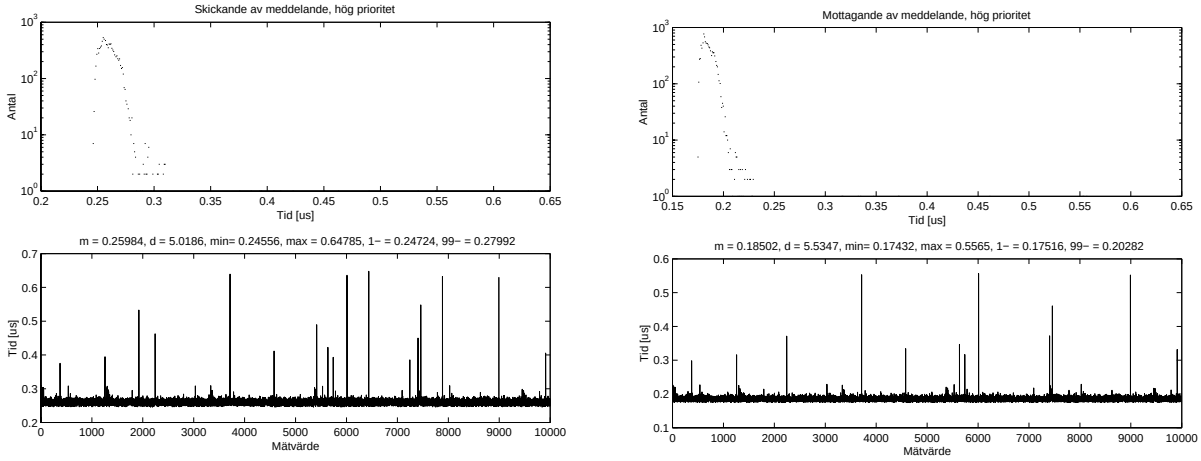
Figur 101 - Hög basprioritet, obelastat system, mottagande tråd har högre prioritet än den sändande



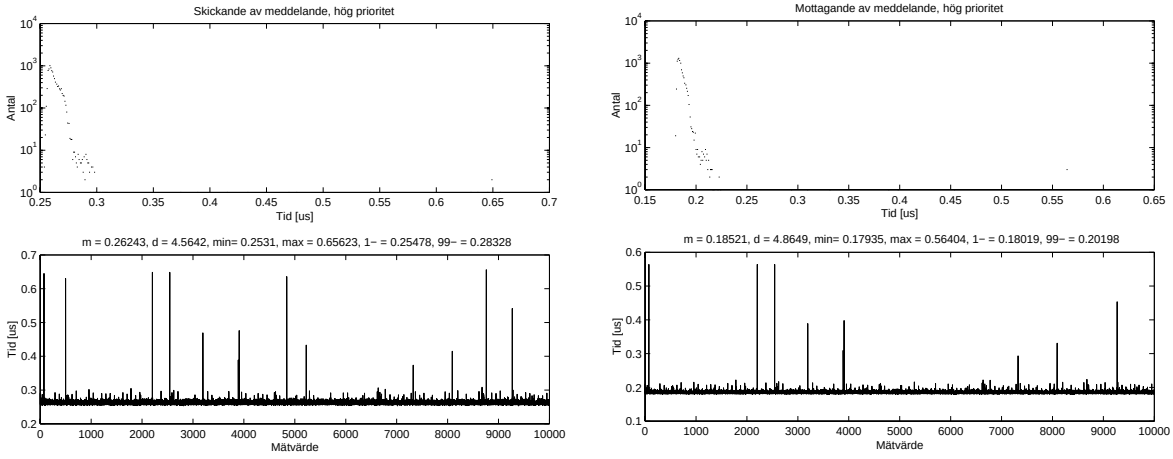
Figur 102 - Realtids basprioritet, obelastat system, mottagande tråd har högre prioritet än den sändande



Figur 103 - Normal basprioritet, belastat system, mottagande tråd har högre prioritet än den sändande

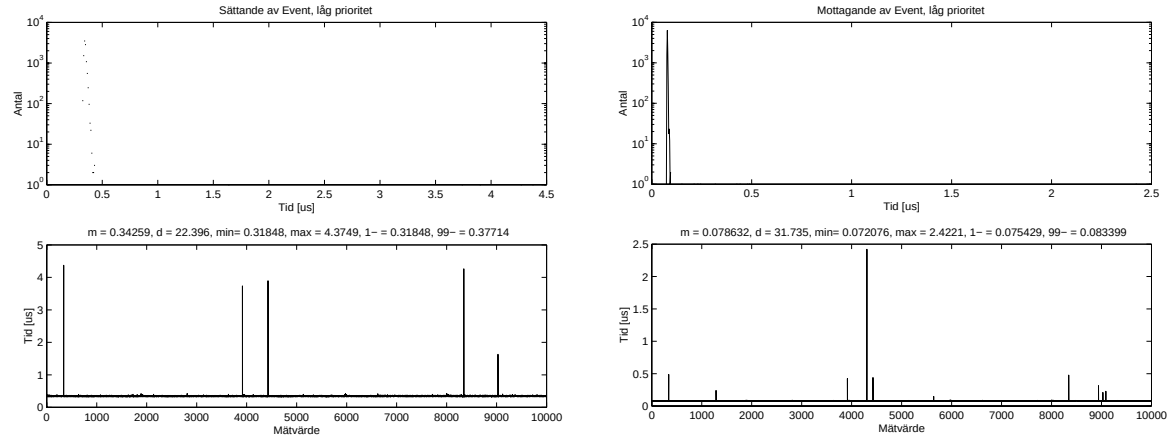


Figur 104 - Hög basprioritet, belastat system, mottagande tråd har högre prioritet än den sändande

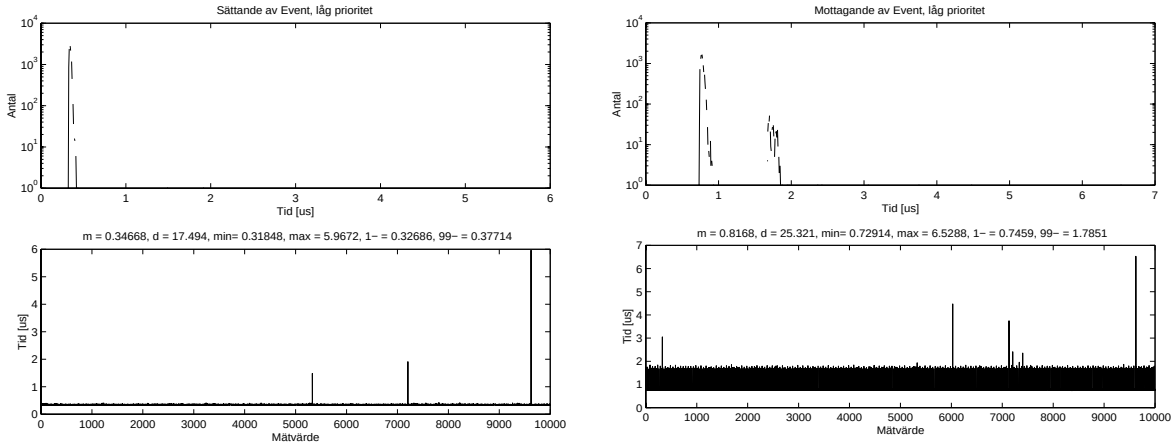


Figur 105 - Realtids basprioritet, belastat system, mottagande tråd har högre prioritet än den sändande

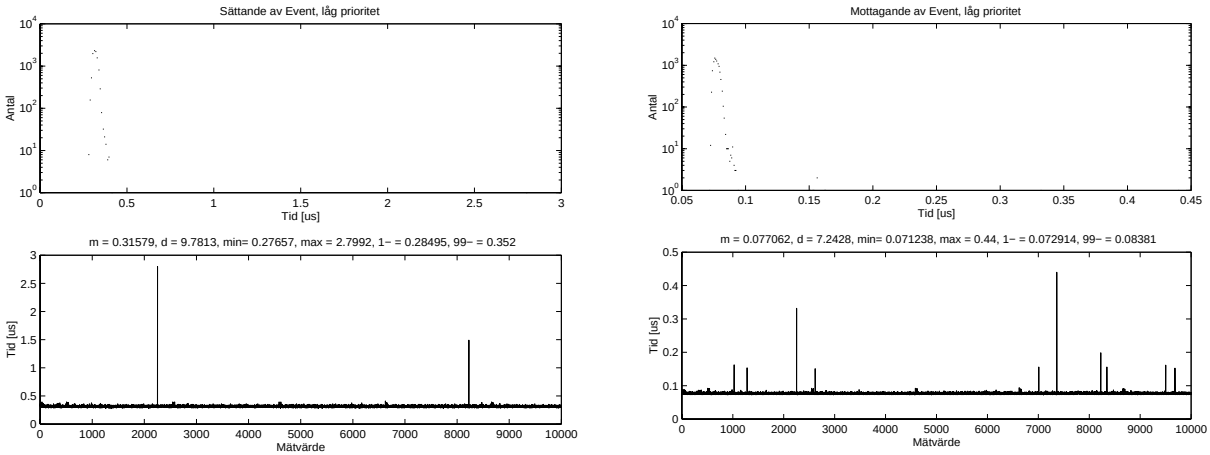
12.1.2 Eventhantering



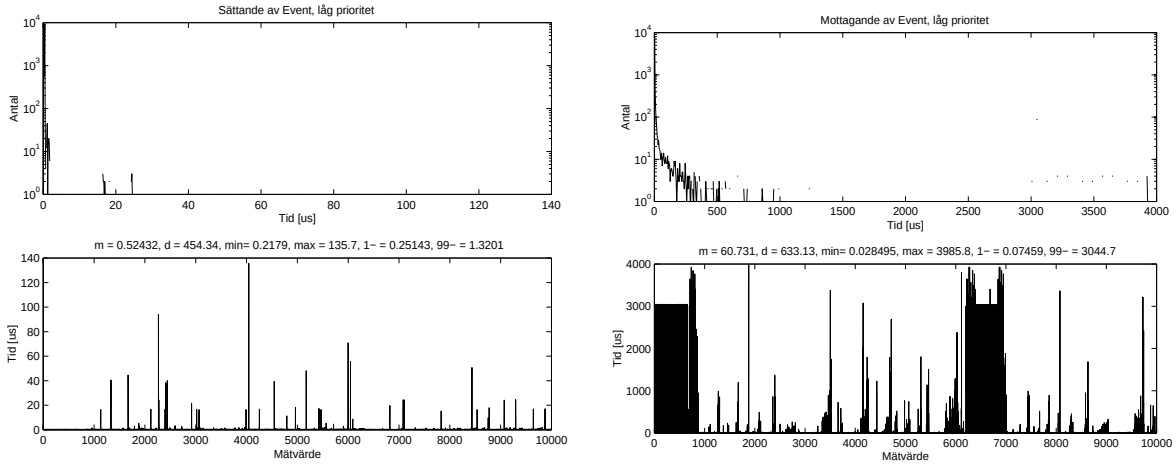
Figur 106 - Normal basprioritet obelastat system, den mottagande tråden har lägre prioritet än den sändande



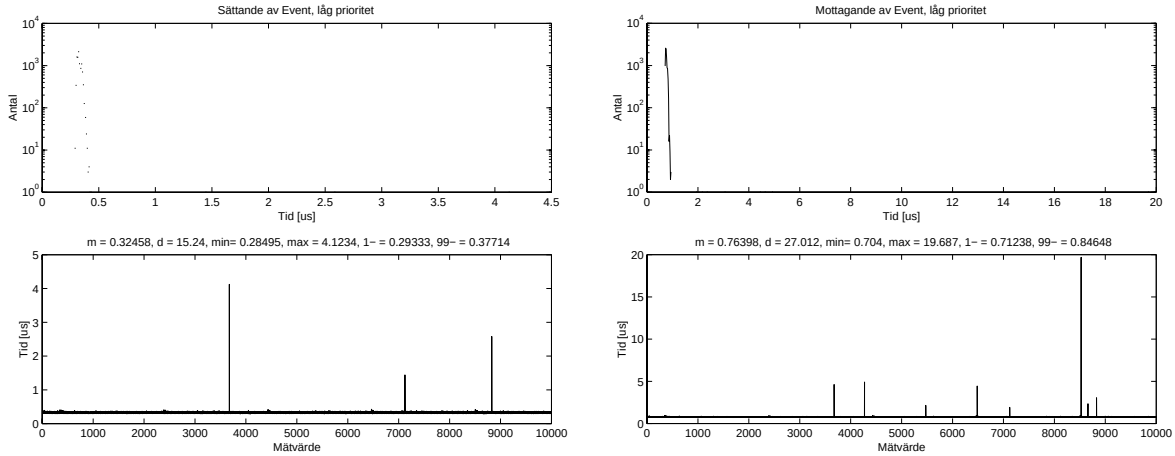
Figur 107 - Hög basprioritet obelastat system, den mottagande tråden har lägre prioritet än den sändande



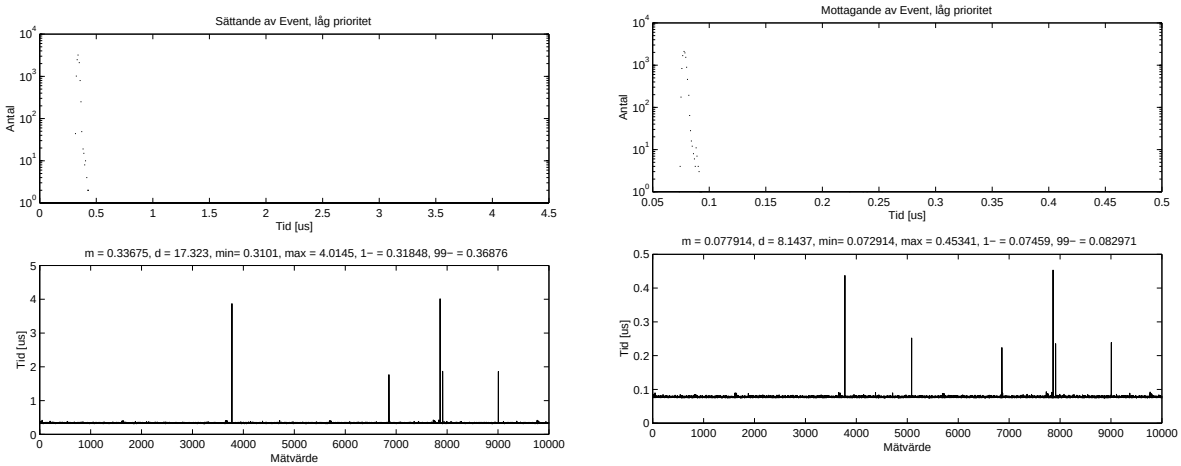
Figur 108 - Realtids basprioritet obelastat system, den mottagande tråden har lägre prioritet än den sändande



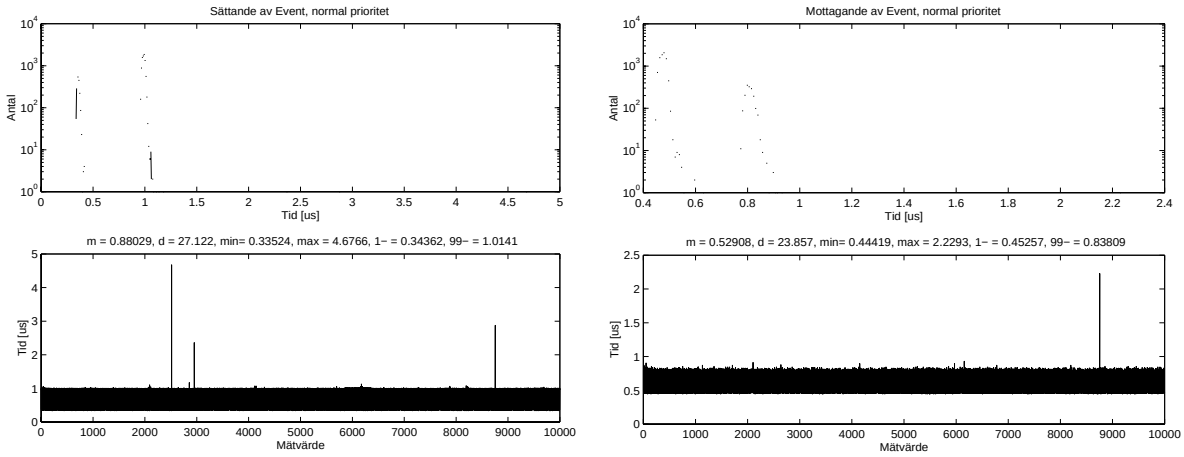
Figur 109 - Normal basprioritet belastat system, den mottagande tråden har lägre prioritet än den sändande



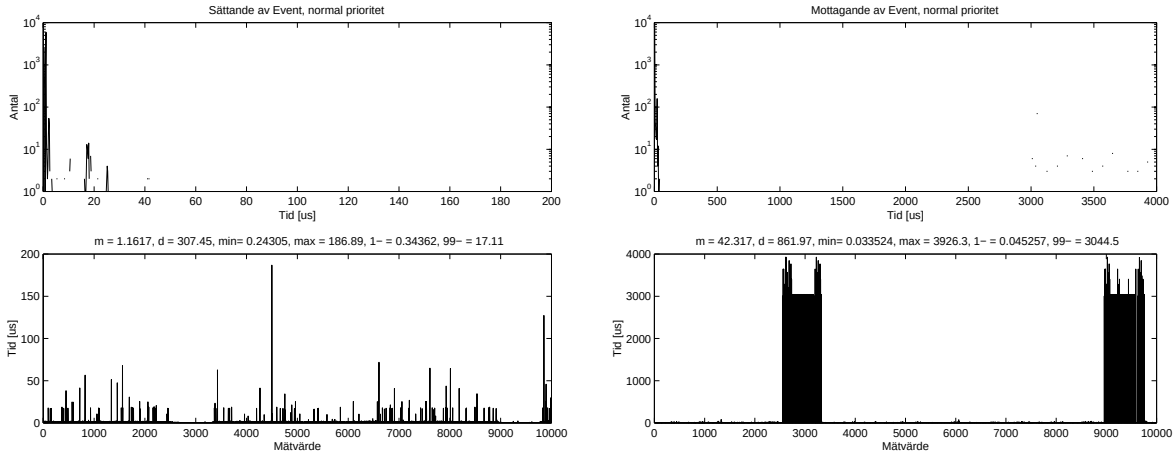
Figur 110 - Hög basprioritet belastat system, den mottagande tråden har lägre prioritet än den sändande



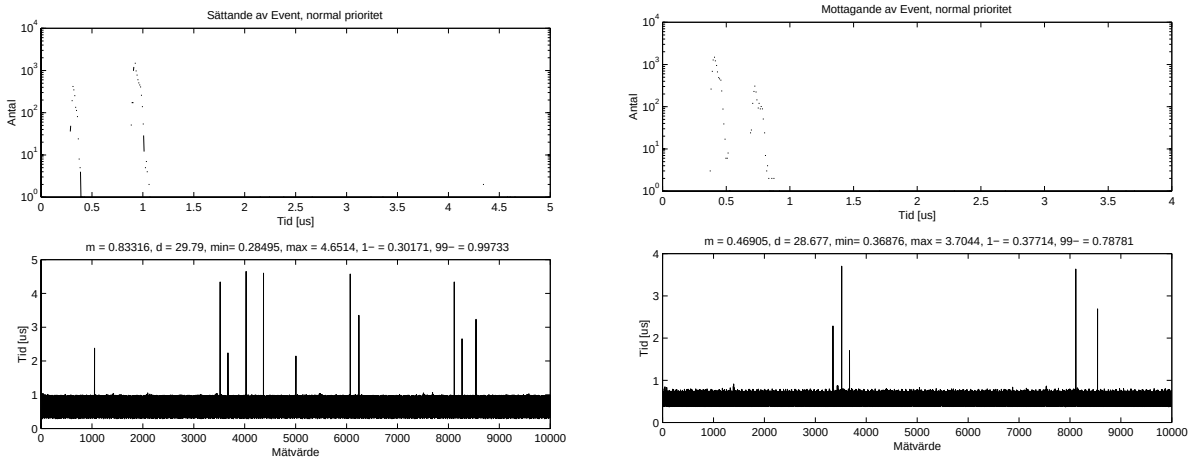
Figur 111 - Realtids basprioritet belastat system, den mottagande tråden har lägre prioritet än den sändande



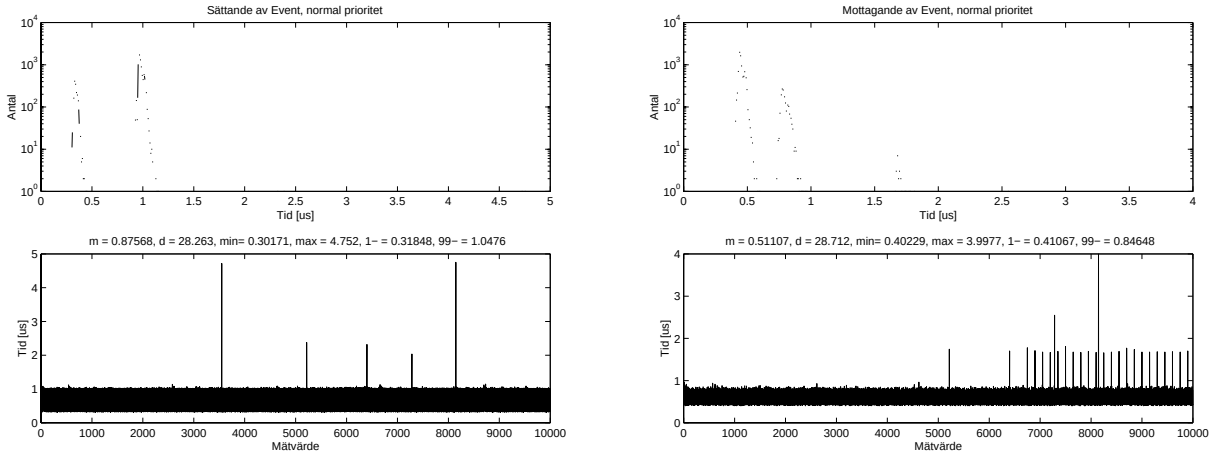
Figur 112 - Normal basprioritet obelastat system, den mottagande tråden har samma prioritet än den sändande



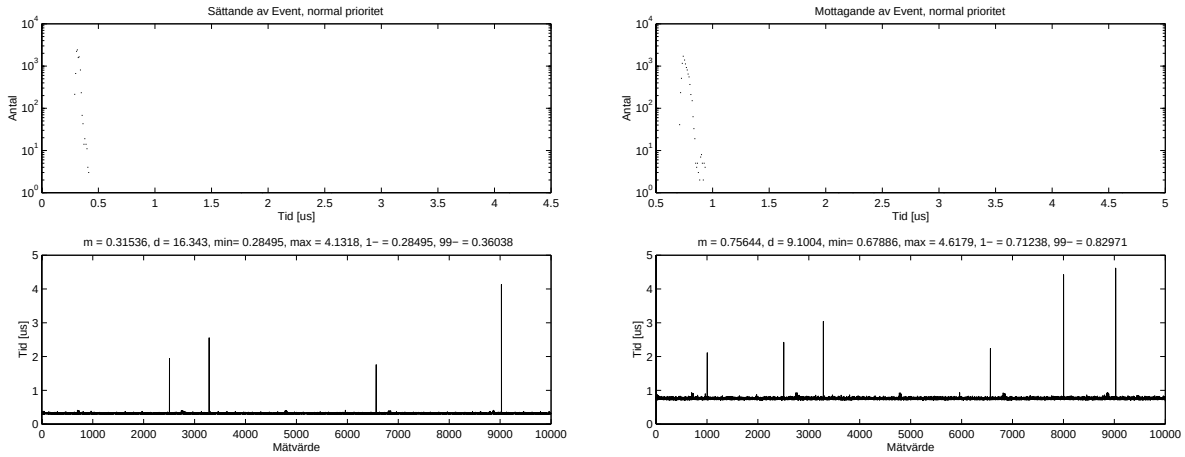
Figur 113 - Normal basprioritet belastat system, den mottagande tråden har samma prioritet än den sändande



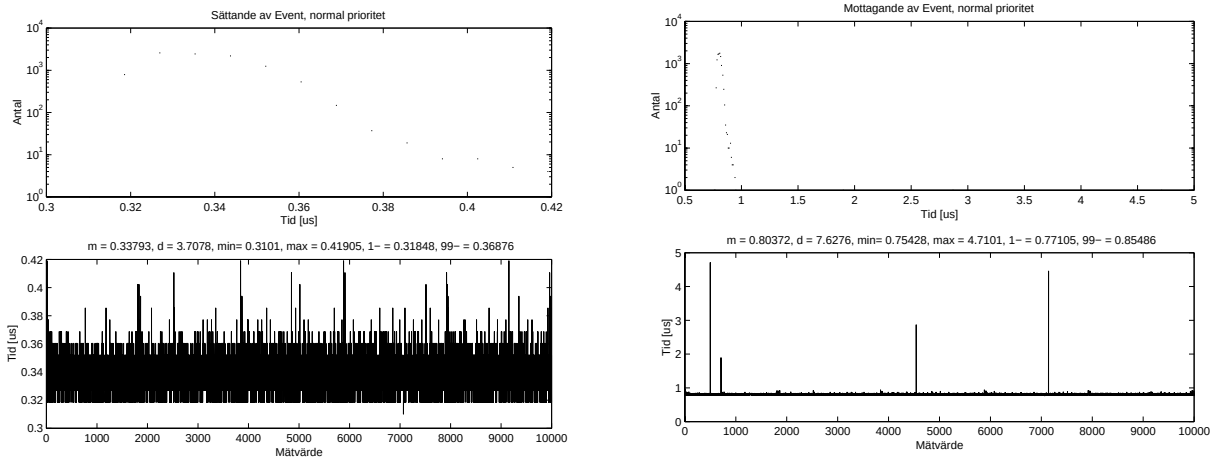
Figur 114 - Hög basprioritet obelastat system, den mottagande tråden har samma prioritet än den sändande



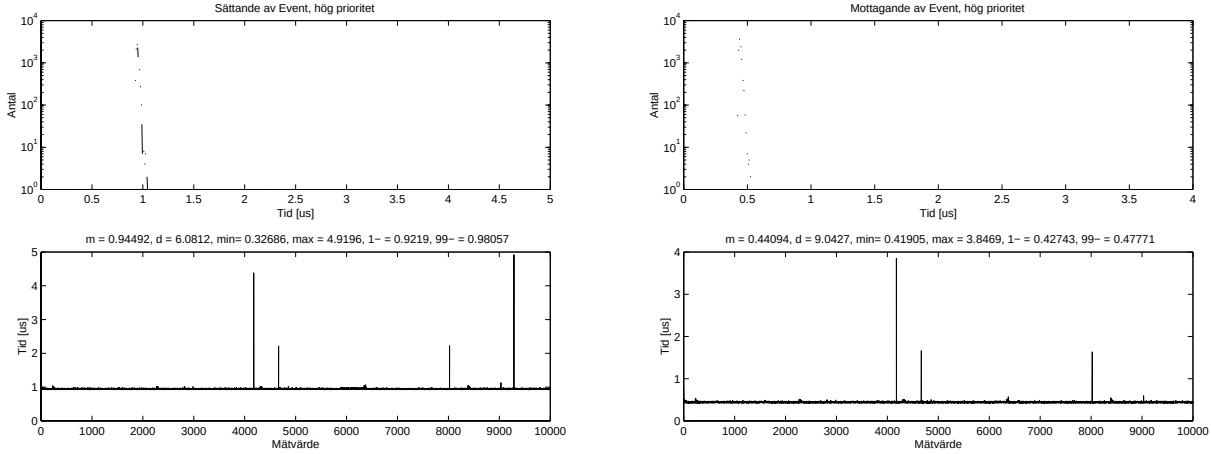
Figur 115 - Hög basprioritet belastat system, den mottagande tråden har samma prioritet än den sändande



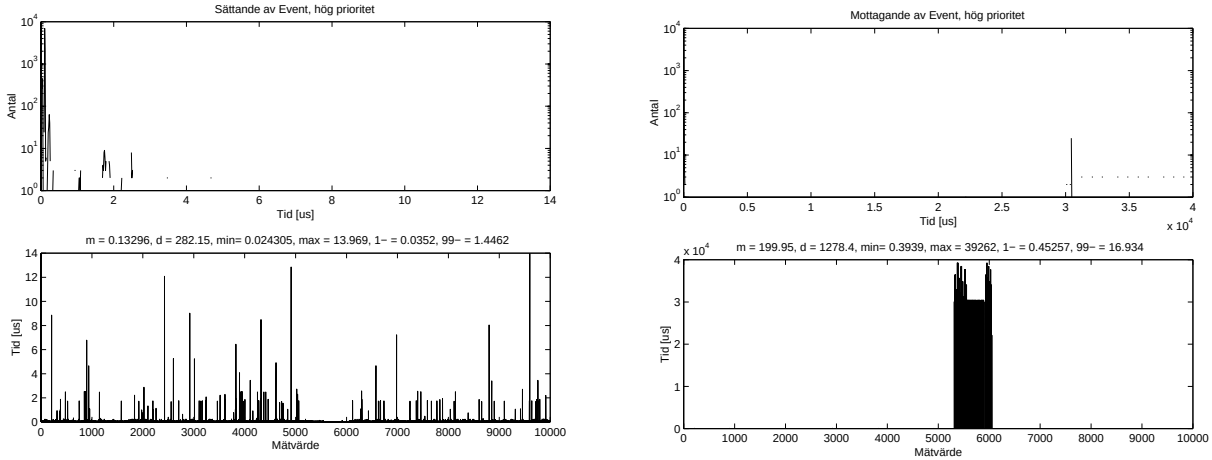
Figur 116 - Realtids basprioritet obelastat system, den mottagande tråden har samma prioritet än den sändande



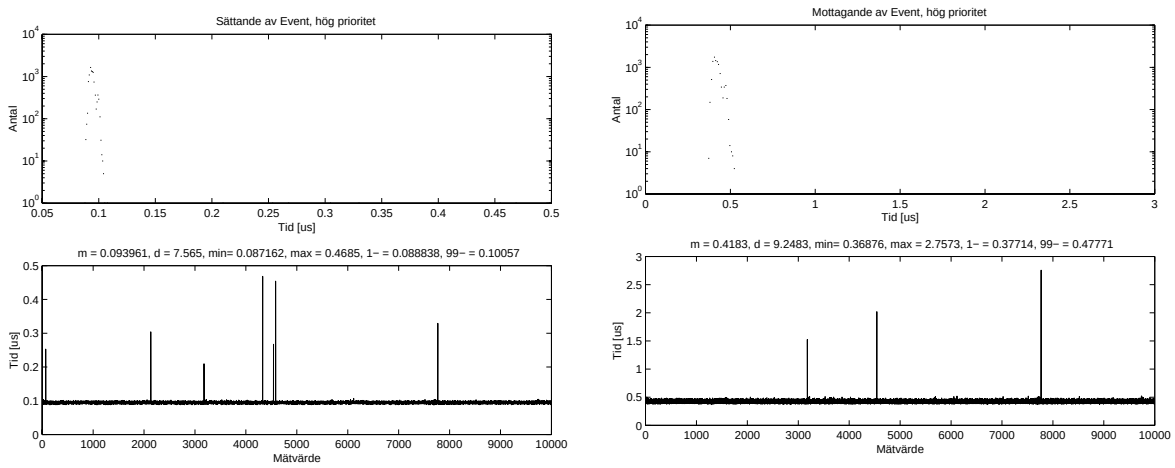
Figur 117 - Realtids basprioritet belastat system, den mottagande tråden har samma prioritet än den sändande



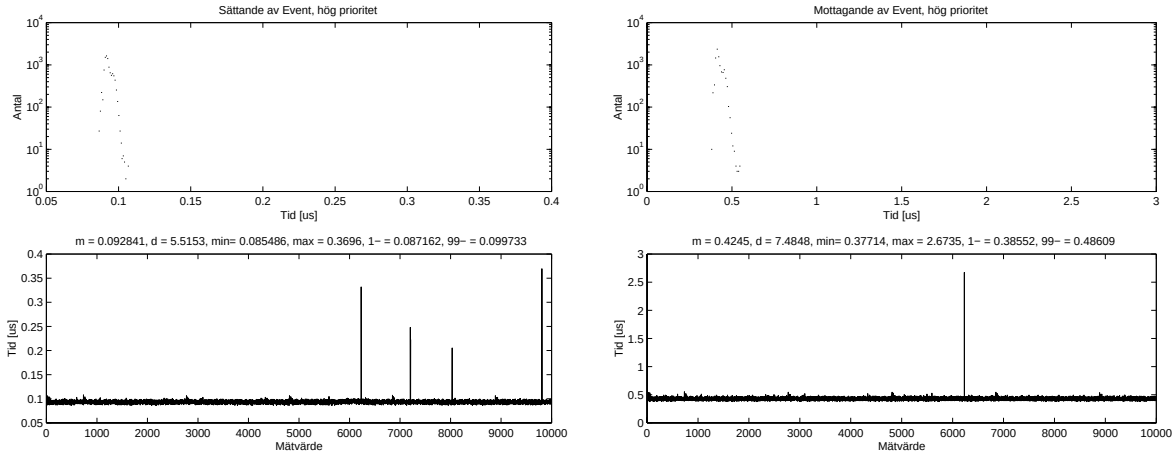
Figur 118 - Normal basprioritet obelastat system, den mottagande tråden har högre prioritet än den sändande



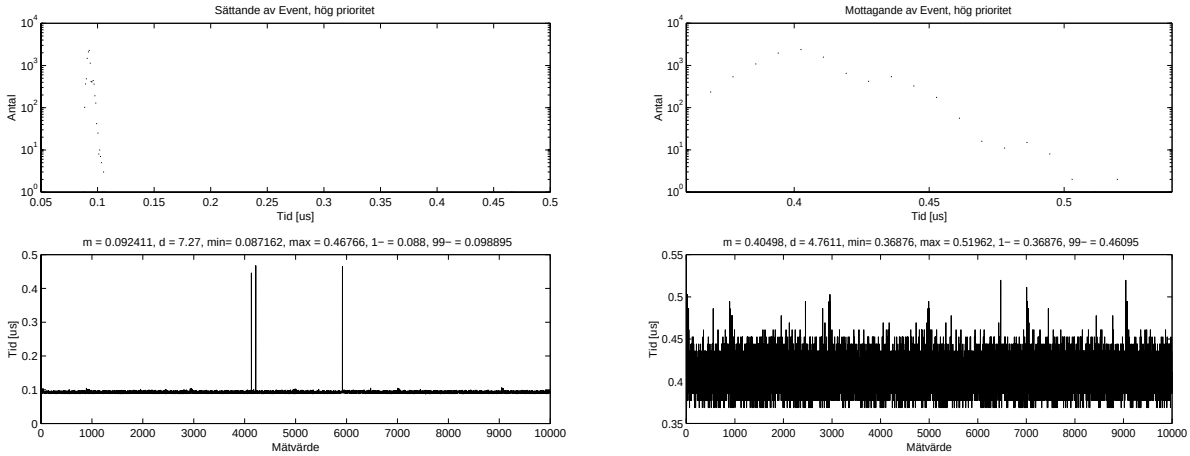
Figur 119 - Normal basprioritet belastat system, den mottagande tråden har högre prioritet än den sändande



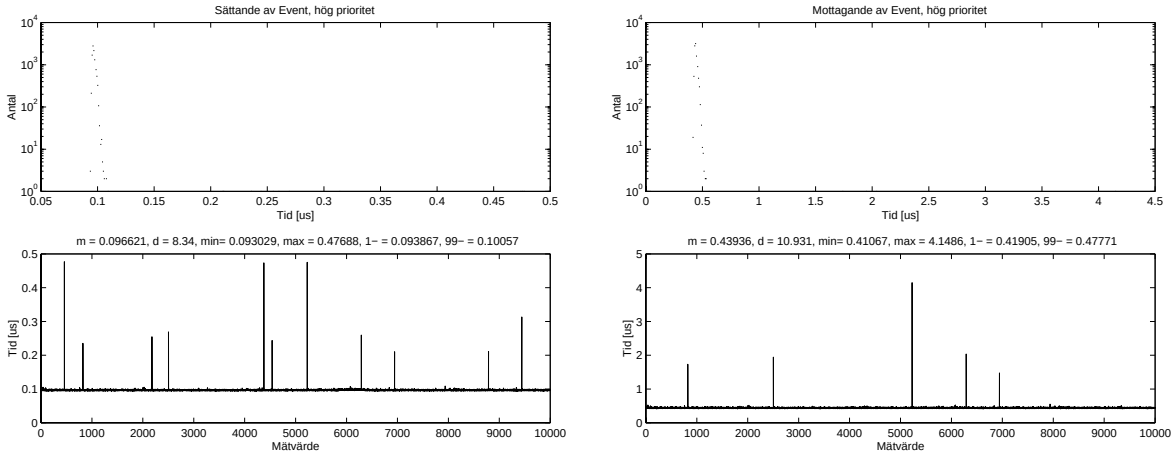
Figur 120 - Hög basprioritet obelastat system, den mottagande tråden har högre prioritet än den sändande



Figur 121 - Hög basprioritet belastat system, den mottagande tråden har högre prioritet än den sändande

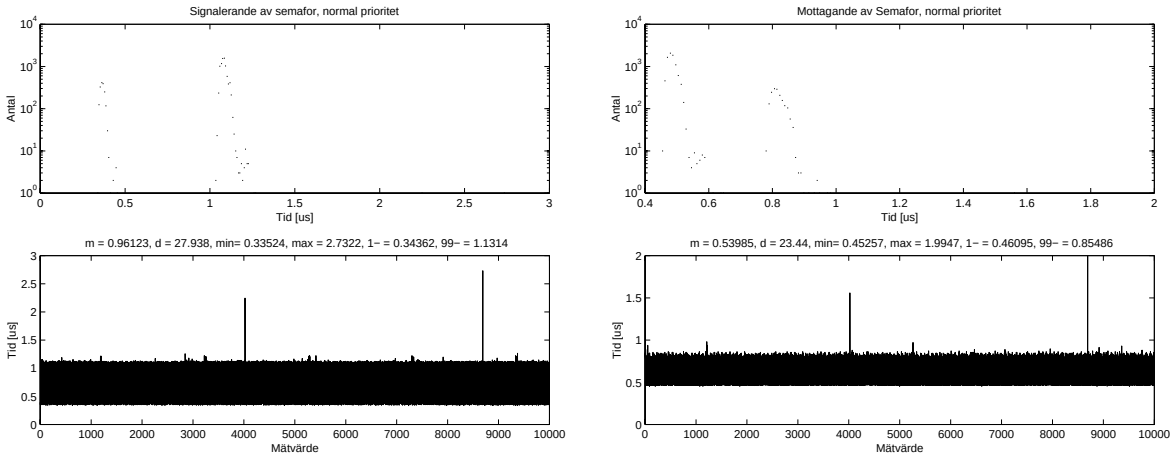


Figur 122 - Realtids basprioritet obelastat system, den mottagande tråden har högre prioritet än den sändande

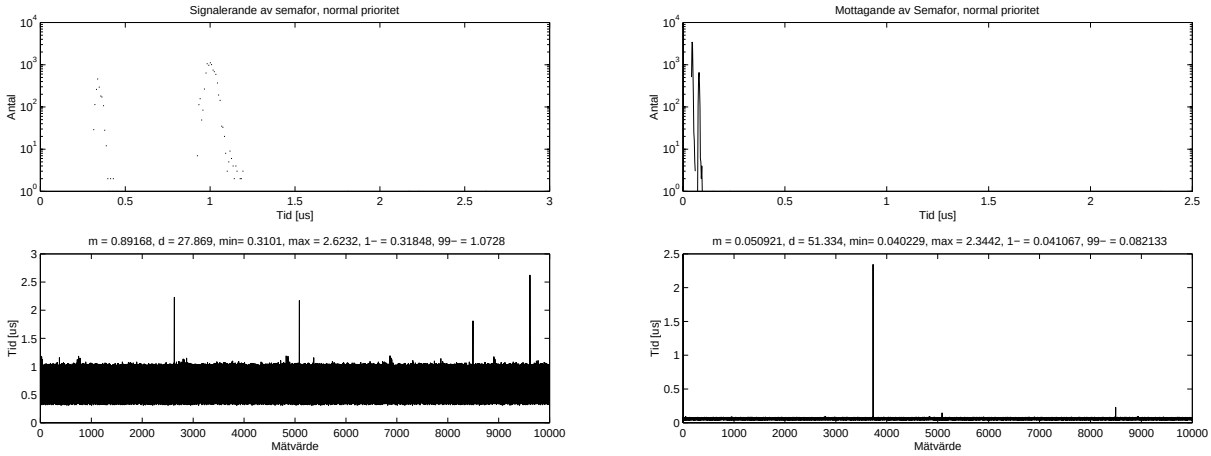


Figur 123 - Realtids basprioritet belastat system, den mottagande tråden har högre prioritet än den sändande

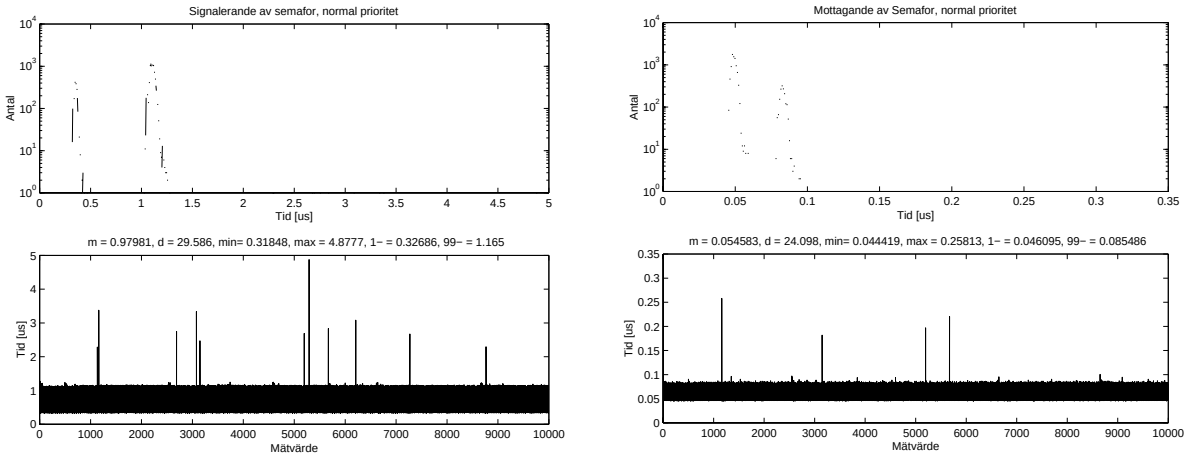
12.1.3 Semaforhantering



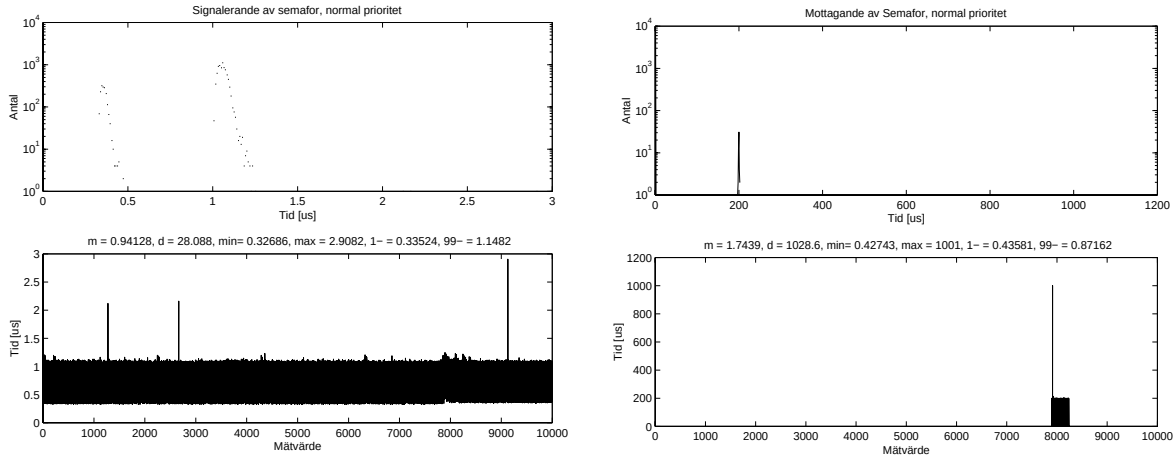
Figur 124 - Normal basklass obelastat system, den mottagande tråden har samma prioritet som den sändande



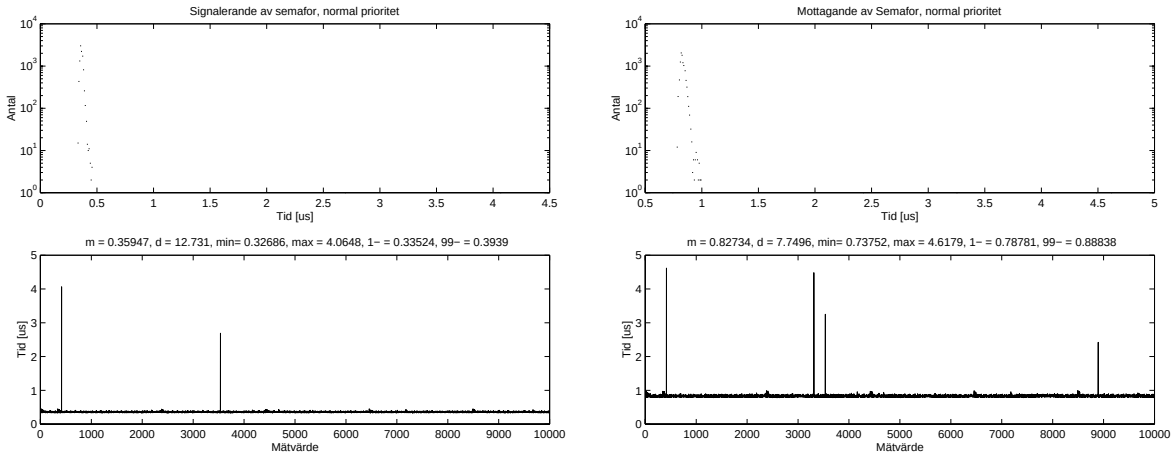
Figur 125 - Normal basklass belastat system, den mottagande tråden har samma prioritet som den sändande



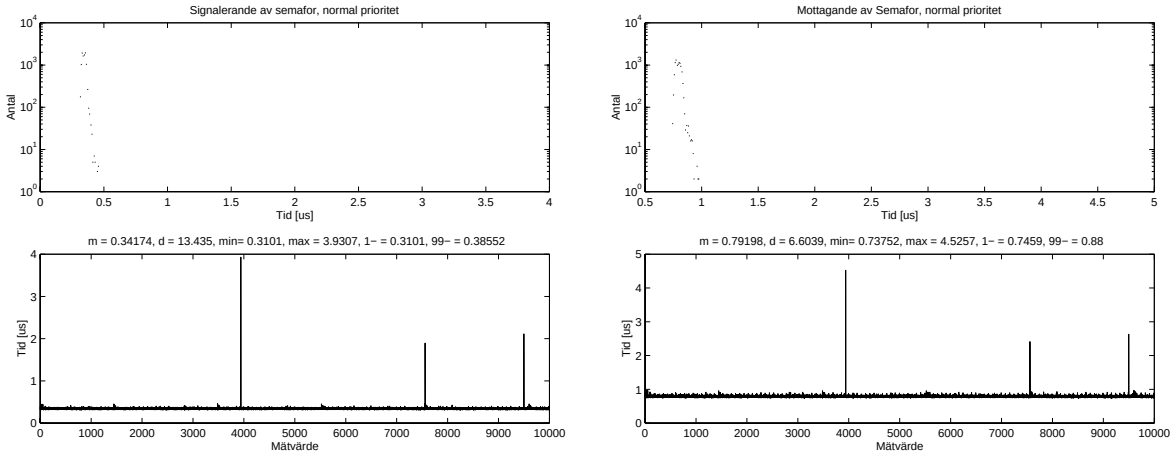
Figur 126 - Hög basklass obelastat system, den mottagande tråden har samma prioritet som den sändande



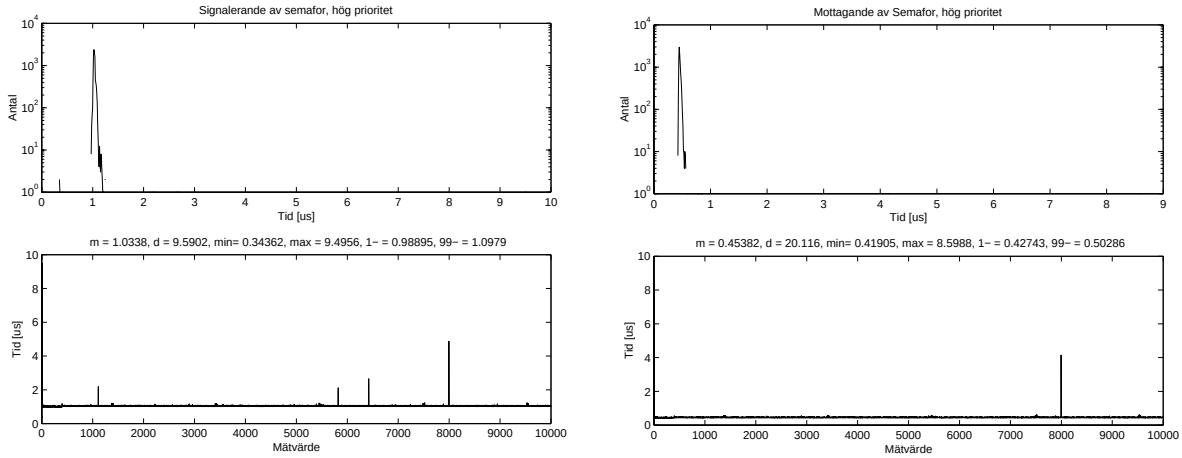
Figur 127 - Hög basklass belastat system, den mottagande tråden har samma prioritet som den sändande



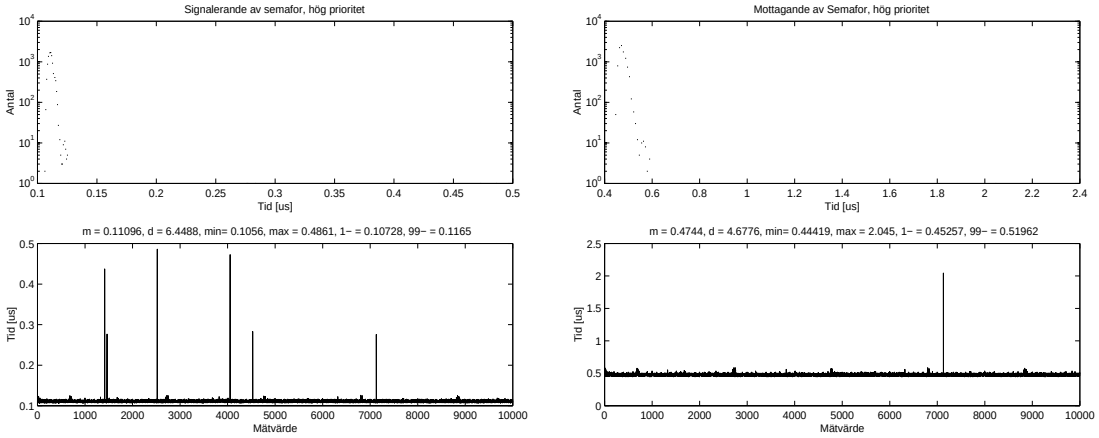
Figur 128 - Realtids basklass obelastat system, den mottagande tråden har samma prioritet som den sändande



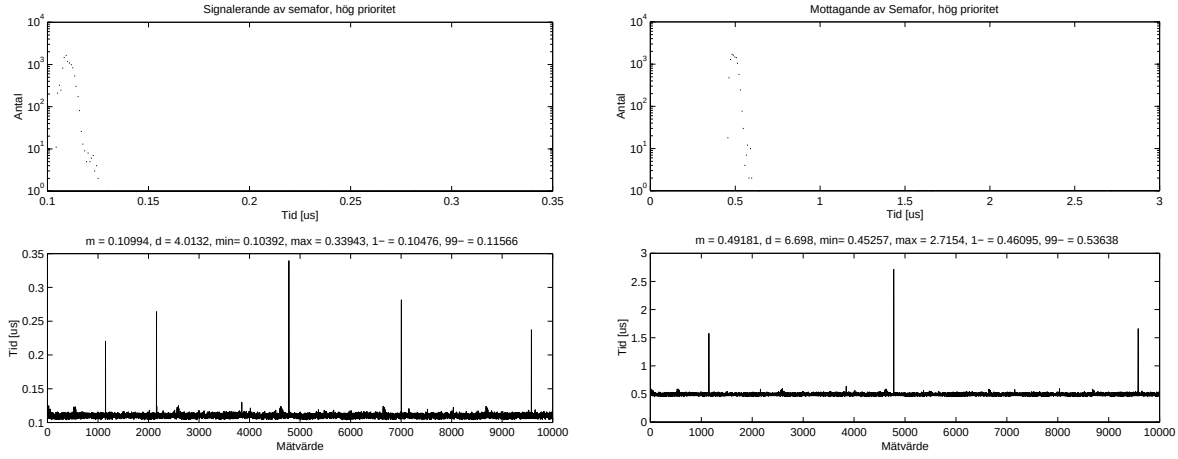
Figur 129 - Realtids basklass belastat system, den mottagande tråden har samma prioritet som den sändande



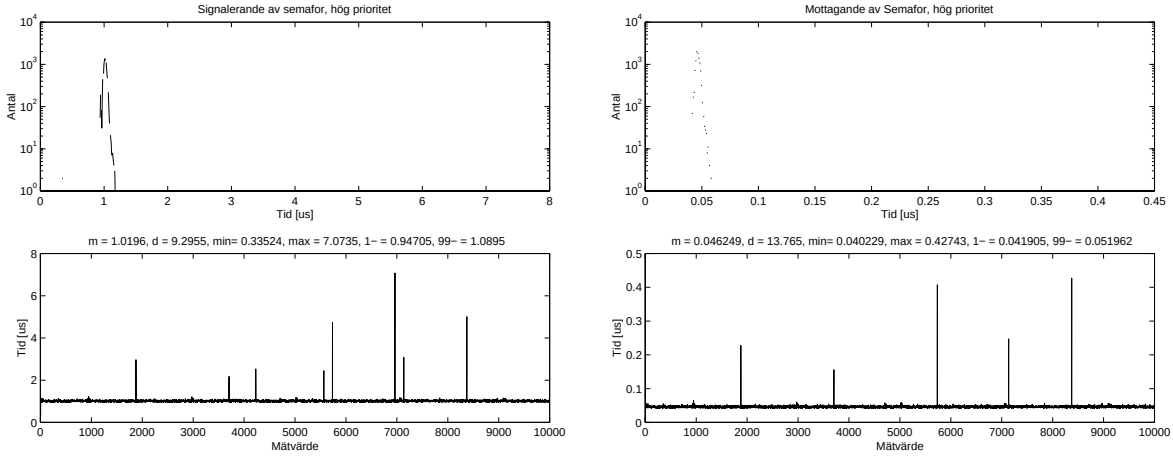
Figur 130 - Normal basklass obelastat system, den mottagande tråden har högre prioritet än den sändande



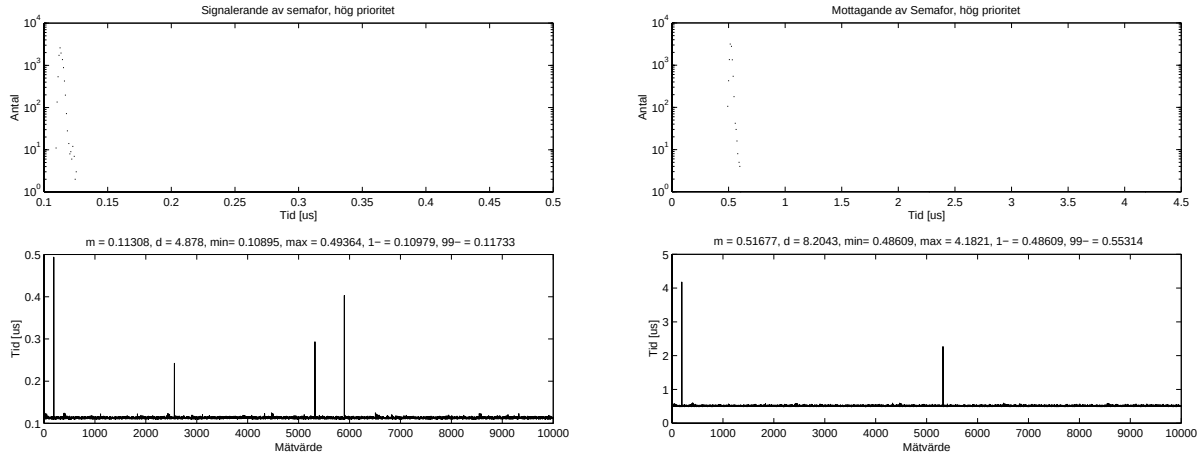
Figur 131 - Hög basklass obelastat system, den mottagande tråden har högre prioritet än den sändande



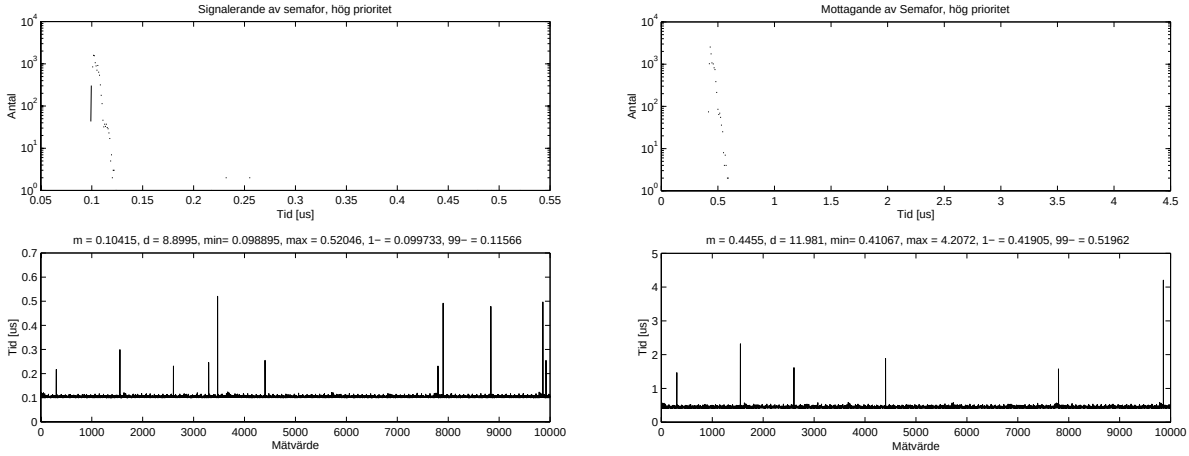
Figur 132 - Realtids basklass obelastat system, den mottagande tråden har högre prioritet än den sändande



Figur 133 - Normal basklass belastat system, den mottagande tråden har högre prioritet än den sändande



Figur 134 - Hög basklass belastat system, den mottagande tråden har högre prioritet än den sändande



Figur 135 - Realtids basklass belastat system, den mottagande tråden har högre prioritet än den sändande

12.2 Systemkonfiguration

Systemkonfigurationen som den detekterades av WinBench

Version	‘98 Build 15
CPU Name	Intel Pentium(R) processor
CPU Family	5

CPU Model	2	
CPU Stepping	4	
CPU Features	0x000001BF	
CPU Clock Speed	75	
CPU L1 Cache (KB)	16	
CPU L2 Cache (KB)	Unknown	
CPU Supports MMX	No	
CPU Floating Point	Yes	
System Name (Make/Model)	Unknown	
System BIOS Information	Unknown	
System BIOS Version	BIOS Version 1.00.03.BS0	
System Bus Type	PCI	
System RAM (MB)	80	
Display Adapter Name (Make/Model)	Cirrus Logic Compatible	
Display Adapter Chip	CL 5430	
Display Adapter DAC	Integrated RAMDAC	
Display Adapter Driver File(s)	File	VersionDescriptionSizeDateTime
	cirrus.sys	4.00.1373.01Miniport-drivrutin för Cirrus-gra-
fik	48752	9/ 3/199612:00 PM
	vga.dll	4.00.1372.01Grafikdrivrutin för 16-färgers
VGA	86480	9/ 3/199612:00 PM
	cirrus.dll	4.00.1372.01Grafikdrivrutin för Cirrus50032
9/ 3/1996	12:00 PM	
	vga256.dll	4.00.1372.01Drivrutin för 256-färgers
VGA\SVGA	52464	9/ 3/199612:00 PM
	vga64K.dll	4.00.1372.01Drivrutin för 32k/64k-färgers
VGA\SVGA	18512	9/ 3/199612:00 PM
Display Adapter Memory (KB)	1024 KB	
Display Adapter BIOS Information	CL-GD543x PCI VGA BIOS Version 1.22 , 01/18/95	
Display Capabilities	Parameter	Value
	Screen Size (pixels)	1024 x 768
	Screen Size (mm)	320 x 240
	Pixels/Inch	96 x 96
	Aspect	36 x 36
	Bits/Plane	8
	Planes	1
	Brushes	2048
	Pens	100
	Fonts	0
	Colors	20
	Palette Size	256
	Reserved	20

	Color Resolution	18
	Clip	1
	Curve	0x01FF
	Line	0x00FE
	Polygon	0x00FF
	Raster	0x7F99
	Text	0x7807

Display Color Reproduction	Unknown	
Display Cursor Type	Unknown	
Display Mode	1024 x 768 8 bits/pixel	
Display Orientation	Landscape	
Display Refresh Rate (Hz)	75	
Display Refresh Mode	Unknown	
Disk Name (Make/Model)	Unknown	
Disk Controller (Make/Model)	Unknown	
Disk Controller RAM (KB)	Unknown	
Disk Windows Cache Type	N/A	
Disk Windows Cache RAM (KB)	Available RAM, favors processes	
Drive Information	Drive	LabelTypeSizeFree Space
	A:\	FAT1423K1261K
	C:\	NTFS1213M919M
	D:\	NTFS1213M
Disk Settings Synchronous buffer commits disabled	N/A	
Disk Settings Write-behind caching for all drives disabled	N/A	
Disk Settings 32 bit protect-mode disk drivers disabled	N/A	
Disk Settings Long name preservation for old programs disabled	N/A	
Disk Settings New file sharing and locking semantics disabled	N/A	
Disk Settings Protect-mode hard disk interrupt handling disabled	N/A	
APM Battery Life	N/A	
APM BIOS Information	Unknown	
APM Enabled	No	
APM AC Power	N/A	
CD-ROM Name (Make/Model)	Unknown	
CD-ROM Controller (Make/Model)	Unknown	
CD-ROM Controller RAM (KB)	Unknown	
CD-ROM Windows Cache Type	Unknown	
CD-ROM Windows Cache RAM (KB)	Unknown	
Sound Adapter Name (Make/Model)	Unknown	
Sound Adapter Driver	Unknown	
Windows Computer Name	SKODA	
Windows Environment Variables	Variable	Value

COMPUTERNAMESKODA
ComSpec C:\WINNT\system32\cmd.exe
HOME c:\ssh
HOMEDRIVE C:
HOMEPATH \
NUMBER_OF_PROCESSORS1
OS Windows_NT
Os2LibPath C:\WINNT\system32\os2\dll;
Path c:\testprogram\bench-marks\winbench98\test\UI32;c:\ssh;C:\WINNT\system32;C:\WINNT
PROCESSOR_ARCHITECTUREx86
PROCESSOR_IDENTIFIERx86 Family 5 Model 2 Stepping 4, GenuineIntel
PROCESSOR_LEVEL5
PROCESSOR_REVISION0204
SystemDrive C:
SystemRoot C:\WINNT
TEMP C:\TEMP
TMP C:\TEMP
USERDOMAINSKODA
USERNAME Administratör
USERPROFILEC:\WINNT\Profiles\Administratör
windir C:\WINNT

Windows Device Drivers File VersionDescriptionSizeDateTime

ntoskrnl.exe 4.00.1381.01NT:s kernel och system863808 9/ 3/1996 12:00 PM

hal.dll 4.00.1372.01DLL för HAL (Hardware Abst-raction Layer) 51424 9/ 3/199612:00 PM

atapi.sys 4.00.1371.01Miniport-drivrutin för ATAPI IDE 18384 9/ 3/1996 12:00 PM

SCSIPORT.SYS4.00.1371.01Drivrutin för SCSI-port31824 9/ 3/1996 12:00 PM

Disk.sys 4.00.1371.01Drivrutin för SCSI-disk14928 9/ 3/1996 12:00 PM

CLASS2.SYS 4.00.1371.01DLL för SCSI Class System 13072 9/ 3/1996 12:00 PM

Ntfs.sys 4.00.1371.01Drivrutin för NTFS366960 9/ 3/ 1996 12:00 PM

Floppy.SYS 4.00.1371.01Drivrutin för diskettenheter18960 9/ 3/1996 12:00 PM

Cdrom.SYS 4.00.1371.01Drivrutin för SCSI-CD-ROM 22128 9/ 3/1996 12:00 PM

Null.SYS 4.00.1371.01NULL-drivrutin2800 9/ 3/1996 12:00 PM

KSecDD.SYS 4.00.1372.01Kernel Security Support Provider Interface 9360 9/ 3/199612:00 PM

Beep.SYS 4.00.1371.01Drivrutin för BEEP4016 9/ 3/ 1996 12:00 PM

sermouse.sys 4.00.1371.01Drivrutin för seriell mus14576 9/ 3/1996 12:00 PM

i8042prt.sys 4.00.1371.01Drivrutin för i8042 Port25168 9/ 3/1996 12:00 PM

mouclass.sys 4.00.1371.01Drivrutin för Mouse Class9360 9/ 3/1996 12:00 PM

kbdclass.sys 4.00.1371.01Drivrutin för Keyboard Class9392 9/ 3/1996 12:00 PM

VIDEOPRT.SYS4.00.1377.01Video Port Driver21744 9/ 3/ 1996 12:00 PM

cirrus.SYS 4.00.1373.01Miniport-drivrutin för Cirrus-gra- fik 48752 9/ 3/199612:00 PM

Msfs.SYS 4.00.1371.01Drivrutin för mailslot22416 9/ 3/ 1996 12:00 PM

Npfs.SYS 4.00.1371.01Drivrutin för NPFS38000 9/ 3/ 1996 12:00 PM

NDIS.SYS 4.00.1371.01Drivrutin för NDIS 3.0 wrapper 120976 9/ 3/1996 12:00 PM

win32k.sys 4.00.1381.01Win32-drivrutin1195472 9/ 3/ 1996 12:00 PM

cirrus.dll 4.00.1372.01Grafikdrivrutin för Cirrus50032 9/ 3/1996 12:00 PM

TDI.SYS 4.00.1371.01TDI Wrapper9712 9/ 3/199612:00 PM

tcpip.sys 4.00.1374.01Drivrutin för TCP/IP133424 9/ 3/ 1996 12:00 PM

netbt.sys 4.00.1377.01Drivrutin för MBT Transport 110928 9/ 3/1996 12:00 PM

elnk3.sys 4.00.1371.01Drivrutin för 3Com Etherlink III 14960 9/ 3/1996 12:00 PM

afd.sys 4.00.1381.01Ancillary Function-drivrutin för WinSock 62480 9/ 3/199612:00 PM

netbios.sys 4.00.1371.01Drivrutin för NetBIOS-gränssnit- tet 29136 9/ 3/199612:00 PM

Parport.SYS 4.00.1371.01Drivrutin för parallellport10096 9/ 3/1996 12:00 PM

Parallel.SYS 4.00.1371.01Drivrutin för parallellportsskriva- re 14896 9/ 3/199612:00 PM

ParVdm.SYS 4.00.1371.01Parallellportsdrivrutin för VDM 6288 9/ 3/1996 12:00 PM

Serial.SYS 4.00.1371.01Seriell drivrutin45104 9/ 3/1996 12:00 PM

Nissm32k.SYS 6.01.03.83NI-DAQ System Settings Manager Driver 39424 10/ 2/199812:02 PM

rdr.sys 4.00.1377.01Drivrutin för filsystemet NT Lan Manager Redirector 257616 9/ 3/199612:00 PM

	Nidaq32k.SYS	6.01.03.83NI-DAQ Windows NT Kernel Dri-
ver	593408	10/ 2/199812:02 PM
	mup.sys	4.00.1379.01Drivrutin för Multiple UNC
Provider	67280	9/ 3/199612:00 PM
	srv.sys	4.00.1381.01Server driver231792 9/ 3/1996
12:00 PM		
	Cdfs.SYS	4.00.1371.01Drivrutin för CD-ROM-filsystem
61296	9/ 3/1996	12:00 PM
	Fastfat.SYS	4.00.1371.01Drivrutin för Fast FAT File Sys-
tem	140048	9/ 3/199612:00 PM
	ntdll.dll	4.00.1376.01DLL för NT Layer354576 9/ 3/
1996	12:00 PM	
	kernel32.dll	4.00.1380.01Windows NT BASE API Client
DLL	365328	9/ 3/199612:00 PM
	wsock32.dll	4.00.1371.01Windows Socket 32-Bit DLL
21264	9/ 3/1996	12:00 PM

Windows Services	Stöd för AFD-nätverksmiljö
	atapi
	Beep
	Cdrom
	cirrus
	Disk
	3Com Etherlink III Adapter Driver
	Floppy
	i8042 tangentbord och drivrutin för PS/2 mus
	Drivrutin för tangentbordklass
	KSecDD
	Mouse Class Driver
	Msfs
	Microsoft NDIS System Driver
	NetBIOS Interface
	WINS Client (TCP/IP)
	Nidaq32k
	NI-DAQ System Settings Driver
	Npfs
	Null
	Parallel
	Parport
	ParVdm
	Serial
	Serial Mouse Driver
	TCP/IP Service

Multimedia Drivers	File	VersionDescriptionSizeDateTime
	mmdrv.dll	4.00.1371.01Drivrutin för MultiMedia Kernel-stöd 16656 9/ 3/199612:00 PM
	msacm32.drv	4.00.00.1371Microsoft Sound Mapper26384 9/ 3/1996 12:00 PM
	midimap.dll	4.00.1371.01MIDI Mapper27408 9/ 3/1996 12:00 PM
	msvidc32.dll	4.00.1371.01Microsoft Video 1-komprimerare 33040 9/ 3/1996 12:00 PM
	iccvid.dll	1.08.00.00Cinepak® Codec77824 9/ 3/1996 12:00 PM
	msrle32.dll	4.00.1371.01Microsoft RLE-komprimerare 14608 9/ 3/1996 12:00 PM
	ir32_32.dll	3.24.15.01Intel Indeo(R) Video R3.2 32-bit Driver 193536 9/ 3/199612:00 PM
	msadp32.acm	4.00.00.1371Microsoft ADPCM CODEC för MSACM 19760 9/ 3/199612:00 PM
	imaadp32.acm	4.00.00.1371IMA ADPCM CODEC för MSACM 21264 9/ 3/199612:00 PM
	msgsm32.acm	4.00.00.1371Microsoft GSM 6.10 Audio CO-DEC för MSACM 27760 9/ 3/199612:00 PM
	tssoft32.acm	1.01.01.05DSP Group TrueSpeech(TM) Audio Codec för MSACM V3.50 12864 9/ 3/199612:00 PM
	msg711.acm	4.00.00.1371Microsoft CCITT G.711 (A-Law and u-Law) CODEC för MSACM 13920 9/ 3/199612:00 PM
	mciwave.dll	4.00.1371.01MCI-drivrutin för waveform-ljud 26896 9/ 3/1996 12:00 PM
	mciseq.dll	4.00.1371.01MCI-drivrutin för MIDI-sequen-cer 25872 9/ 3/199612:00 PM
	mcicda.dll	4.00.1373.01MCI-drivrutin för cdaudio-enhe-ter 21776 9/ 3/199612:00 PM
	mciavi32.dll	4.00.1371.01MCI-drivrutin för Video For Win-dows 90384 9/ 3/199612:00 PM
	ddraw.dll	4.00.1374.01Direct Draw138000 9/ 3/1996 12:00 PM
	dsound.dll	4.00.1371.01DirectSound91408 9/ 3/1996 12:00 PM
	dplay.dll	4.00.1371.01Microsoft DirectPlay28944 9/ 3/ 1996 12:00 PM
Windows Version	Windows NT 4.0, Build 1381	
Windows System Metrics	Metric	Value
	Boot	0
	DBCS	0
	Debug	0
	Network	3
	Secure	0

	Border	1 x 1
	Cursor	32 x 32
	Dialog Frame	3 x 3
	Double-click	4 x 4
	Frame	4 x 4
	Display Resolution	1024 x 768
	Display Client Area	1024 x 749
	H Scroll	16 x 16
	V Scroll	16 x 16
	H Thumb Width	16
	V Thumb Height	16
	Icon	32 x 32
	Small Icon	16 x 16
	Icon Spacing	82 x 64
	Maximized	1032 x 776
	Minimized	160 x 24
	Minimized Spacing	160 x 24
	Max Track	1036 x 780
	Min Track	112 x 27
	Caption Height	19
	Menu Height	19
	Menu Drop Alignment	0
	Mouse Present	1
	Swap Button	0
	Kanji Window	0
	Mid-East	0
	Pen Extensions	0
	Show Sounds	0
	Slow Machine	0

Windows System Parameters	Parameter	Value
	Beep Active	Yes
	Border Multiplier	1
	Default Input Language	0000041D
	Drag Full Windows	Yes
	Font Smoothing	No
	Grid Granularity	Unknown
	Mouse Trails	Unknown
	Low Power Active	Unknown
	Low Power Timeout	Unknown
	Power Off Active	Unknown
	Power Off Timeout	Unknown

Prefers Keyboard	Unknown
Screen Reader	Unknown
Screen Saver Active	No
Screen Saver Timeout (sec)	900
Windows Extension	Unknown

Display Adapter Driver Acceleration	N/A
Display Refresh Pattern	Unknown
Disk Settings Read Ahead Threshold	N/A
Disk Settings Path Cache	N/A
Disk Settings Name Cache	N/A
Disk Settings CDFS Prefetch	N/A
Disk Settings CDFS Prefetch Tail	N/A
CPU Active Processors	1