En jämförelse av scheduleringsalgoritmer lämpade för realtidssystem

Ville-Matti Saarelainen 1800137
Kandidatavhandling i datavetenskap
Handledare: Andreas Lundell
Fakulteten för naturvetenskaper och teknik
Åbo Akademi
2022

Innehåll

1 Referat

Realtidssystem är datorsystem där det är kritiskt att operationer sker inom utsatt tid. Det klassiska exemplet är system som styr maskiner, exempelvis bromsarna i en bil eller en robotarm, men även latenskritisk apparatur såsom nätverksapparatur och multimediasystem kan behandlas som realtidssystem.

I realtidssystem definieras en deadline för alla processer som kräver realtidsprecision i sin exekvering. Oftast kommer deadlinen uppenbarligen från den fysiska världen. Exempelvis ifall en sensor skickar ett mätvärde med jämna mellanrum, vill man garantera att processen som lagrar mätvärdet alltid har exekverat färdigt förrän nästa mätvärde kommer. Ifall det inte lyckas kunde mätvärden tappas bort eller kunde systemet fastna med en växande kö av mätvärden.

För att realtidssystemets alla processer garanterat ska möta sina deadlines måste även scheduleraren i operativsystemet anpassas ändamålsenligt. Det är även viktigt att det går att resonera kring schedulerarens beteende, speciellt i värsta fall där möjligheten för att systemet misslyckas i scheduleringen är störst.

I motsats till mer traditionella schedulerare betonar realtidsschedulerare processernas exekveringstid och deras deadliner mer än systemets totala effektivitet eller schedulerarens responstid[3]. Orsaken är att realtidsschedulerare trots sitt namn inte kräver bra prestanda. Det ända de kräver är att det går att formellt bevisa att systemet även i värsta fall klarar av att utföra alla uppgifter inom utsatt tid. Givetvis är optimering av realtidsschedulerarna även ett forskningsområde i sig. I denna avhandling presenteras metoder för att klassificera och analysera realtidssystem, samt ett urval av scheduleringsalgoritmer som kan tillämpas inom realtidssystem och realtidsoperativsystem.

2 Introduktion

Teknikens utveckling har skapat och fortsätter skapa allt större och sammanlänkade datorsystem. Däremot utvecklas inte batteriernas kapacitet eller antenners sändstyrka per energienhet med samma takt, vilket ställer allt högre prestandakrav på system med avsevärt begränsade resurser, såsom sensorer och inbyggda system som styr fysiska apparater. I allt större utsträckning förväntas kritiska inbyggda system rapportera regelbundet över datornätverk och rita upp grafiska användargränssnitt utöver att garantera att maskinen eller apparaturen styrs säkert och med absolut precision.

För att hålla utvecklingen av system som förväntas utföra alla dessa uppgifter samtidigt, måste systemen splittras upp i flera processer. Operativsystemet, i den utsträckning ett sådant finns på plattformen i fråga, måste därmed köra alla dessa processer så att det kan garantera att de viktigaste processerna alltid körs till slut inom utsatt tid.

Exempelvis måste en bil genast bromsa då föraren trycker in bromspedalen, inte fortsätta rulla framåt i en obestämd tid eftersom styrenheten i bilen fortfarande försöker hitta den bästa rutten till destinationen. Ej heller kan processer med låg prioritet ignoreras. Förr eller senare förväntar sig chauffören att navigationssystemet har arbetat fram en rutt, även om det är acceptabelt att det tar några sekunder. Scheduleringsalgoritmen som används i bilens inbyggda dator måste således se till att bilen alltid bromsar så snabbt som möjligt när det krävs, men den måste även tillåta exekvering av mindre viktiga processer då och då. I denna avhandling presenteras de vanligaste scheduleringsalgoritmerna,

3 Bakgrund

3.1 Scheduleringsproblemet

Frågan om hur man ska schedulera en begränsad mängd resurser på bästa möjliga sätt har studerats i mer än 50 år, och har tillämpningar bl.a. inom företagsledning, maskinanvändning i industrianläggningar, och naturligtvis datavetenskap [7]. Vilken den optimala uppdelningen av resurserna är beror på målen man vill uppnå genom scheduleringen, dvs. vad man vill optimera.

I den här avhandlingen behandlas det mer specifika problemet av resursanvändning på processorer i datorer. En processor kan nämligen endast utföra ett begränsat antal beräkningar per tidsenhet, men så gott som alltid krävs det att flera processer delar på beräkningskapaciteten för att få en fungerande applikation. Av speciellt intresse är hur man kan analysera resursanvändningen av ett givet system och garantera att alla processer kommer att exekvera inom utsatt tid.

3.2 Processer

E. W. Dijkstra beskrev redan år 1968 i sina föreläsningsanteckningar [5] hur ett flertal processer kan exekveras parallellt på en processor, samt hur exekveringsordningen inte är definierad. Därmed kan såväl körtiden som tidspunkten då processen kört färdigt för en given process variera. Denna grund är relevant ännu idag, såväl för traditionella som realtidsprocesser.

diagram från [3] s. 65

3.3 Scheduleringsalgoritmer

Varje gång en process exekverats färdigt eller avbryts, måste processorn avgöra vilken process som bör exekveras till näst. Beslutet fattas av en scheduleringsalgoritm[3], på basis av informationen som finns att tillgå om varje process i systemet. En väldigt enkel scheduleringsalgoritm kunde exekvera varje process i systemet i tur och ordning (eng. Round Robin), men det leder inte alltid till önskade resultat. Ifall operativsystemet har en stor mängd bakgrundsprocesser och en video-avkodningsprocess, kommer systemet spendera mycket tid på bakgrundsprocesserna och för lite på avkodningsprocessen. Då blir videouppspelningen hackig. En mycket komplex algoritm kan använda sig av en stor mängd information som operativsystemet lagrar om varje process, men då kräver algoritmen mer beräkningskapacitet, vilket inte heller alltid är önskvärt. Denna aspekt granskas närmare i X.Y.Z.

kap.num

3.4 Traditionella schedulerare

Med traditionella eller icke-realtidsschedulerare avses schedulerare i ”vanliga” datorer, exempelvis bordsdatorer i ett kontor. I stort sett är denna typ av datorer inte bekymrade över hur länge det tar att köra olika program, eftersom det inte är katastrofalt ifall en process ibland exekveras efter 5ms och ibland efter 10ms. Till en viss grad kan det även accepteras att någon process blockerar andra processer från att köra.

3.4.1 Målsättningar för traditionella schedulerare

I sin bok nämner Bertolotti et al. [3] tre exempel på aspekter som traditionella schedulerare ofta betonar. Den första är rättvishet, dvs. att alla processer i systemet bör få en chans att exekvera. I praktiken innebär det att en mindre viktig process kan väljas för exekvering ifall den väntat länge på att få exekvera. Nackdelen med en sådan schedulerare är att det blir svårare att förutspå hur systemet kommer att bete sig, vilket är en viktig egenskap för realtidsschedulerare.

Den andra är effektivitet, med vilket menas att själva scheduleringsalgoritmen inte förbrukar för mycket av beräkningskapaciteten. Eftersom en vanlig bordsdator kan förväntas köra hundratals processer samtidigt, är nästan alla traditionella scheduleringsalgoritmer av komplexitetsklassen O(1), eller åtminstone inte beroende av det totala antalet processer i systemet[3].

Den tredje aspekten är genomströmning (eng. throughput), vilket är viktigt exempelvis i datorkluster där man önskar köra många jobb så snabbt och energieffektivt som möjligt. Schedulerare som betonar genomströmningen av processer i systemet försöker få scheduleraren att köra så snabbt som möjligt, så att så mycket som möjligt av processortiden används till det verkliga beräknandet.

Problemet med dessa schedulerare ur realtidssystemets synvinkel är att man svårligen kan uttala sig om exakt hur länge man måste vänta innan en viss process körs. Ifall en process måste exekveras färdigt inom ett strikt tidsintervall, kan det hända att kravet möts eller inte möts beroende på schedulerarens beteende givet ett flertal processer i systemet. Problematiken är nära besläktad med synkroniseringsproblemen och kapplöpningsproblemen som är vanliga inom parallellprogrammering, där vissa exekveringsordningar ger riktiga resultat och andra inte [3].

4 Realtidssystem

Realtidssystem är datorsystem där det är kritiskt att operationer sker inom utsatt tid. Det klassiska exemplet är system som styr maskiner, exempelvis bromsarna i en bil eller en ro-

botarm, men även latenskritisk apparatur såsom nätverksapparatur och multimediasystem kan behandlas som realtidssystem [1]. I samband med schedulering menas med realtidssystem ett system av processer med tidsmässiga krav på när processerna i systemet exekverats färdigt. Eftersom realtidssystem av sin natur ofta kontrollerar maskiner och annan kritisk apparatur är det även av intresse att kunna formellt bevisa att systemet fungerar korrekt och inte utgör en säkerhetsrisk.

4.1 Realtidsprocesser

Teorin för realtidsprocesser bygger på den som presenterats i avsnitt 3.2. Enligt Abeni m. fl.[1] har en realtidsprocess två definierande egenskaper: processens deadline samt dess värsta-falls exekveringstid. Dessutom måste en process släppas (eng. be released), köas eller på annat vis bli klar för exekvering vid någon tidpunkt för att överhuvudtaget utgöra en del av ett större system av processer. Dessa egenskaper utgör grunden för analys av realtidssystem.

4.1.1 Exekveringstid

Varje process kräver en viss tid för att exekvera. Exekveringstiden definieras som tidsintervallet från att processen börjar exekveras av processorn tills den kört till slut, givet att den inte störs av externa faktorer såsom andra processer i systemet [3]. De flesta processer har dock flera exekveringsstigar, och exekverar olika länge beroende på indatan de får. Mängden tid varje process kräver att exekveras inverkar naturligtvis direkt på huruvida processen möter sin deadline eller inte. Scheduleraren kan inte i förväg veta den exakta tiden en sådan process kräver att exekveras, och därmed måste man utgå från att varje process alltid kräver värsta-falls exekveringstiden [3].

För vissa typer av processer kan det vara svårt eller omöjligt att definiera en rimlig värsta-falls exekveringstid[4][3]. Minnesintensiva och nätverksintensiva processer är vanliga exempel på sådana. Dessa processers exekveringstid är alltså beroende av operationer vars tidskrav är oförutsägbara, och således kan deras värsta-falls exekveringstid vara avsevärt långa eller rentav odefinierade. För att analysera sådana processer är den enda möjligheten att göra en uppskattning på en rimlig men inte strikt värsta-falls exekveringstid istället. Detta behandlas närmare i kapitel 8.8

4.1.2 Deadliner

I realtidssystem har varje process som kräver realtidsprecision i sin exekvering en väldefinierad deadline [3]. Oftast kommer deadlinen uppenbarligen från systemet själv eller apparaturen som systemet kontrollerar. Om en sensor skickar ett mätvärde med jämna

mellanrum, vill man garantera att processen som lagrar mätvärdet alltid har exekverat färdigt förrän nästa mätvärde kommer. Ifall det inte lyckas kunde mätvärden tappas bort eller kunde systemet fastna med en växande kö av mätvärden.

Deadliner kan vara antingen absoluta eller relativa [3]. Ett exempel på en absolut deadline är ett mätvärde som uppmäts varje sekund, oberoende när det förra värdet lagrades. En relativ deadline kommer en viss tid efter att processen släpps. Det kunde exempelvis vara en signal att bromsa som måste behandlas inom en bråkdels sekund när processorn får signalen.

4.1.3 Periodicitet

Vissa arbetsuppgifter tenderar vara periodiska, med följden att processerna som utför de arbetsuppgifterna måste exekveras med jämna mellanrum. Realtidssystem som styr eller stabiliserar apparatur med hjälp av mätvärden och modellering är ett exempel på processer som kör i regelbundna cykler.

Realtidssystem bestående av endast periodiska processer kan scheduleras genom förbestämda exekveringsscheman [3], dvs. genom offline-schedulering. Denna typ av schedulering kommer behandlas i kapitel 7.

4.2 Klassificering av realtidssystem

Realtidssystem klassificeras oftast som hårda eller mjuka beroende på hur kritiska dess uppgifter är. I vissa system, såsom bromssystemet i en bil som beskrevs i kapitel 2, är det uppenbarligt att allvarliga säkerhetsrisker uppstår ifall systemet inte lyckas uppnå sina deadliner. Dessa system kallas hårda realtidssystem (eng. hard real-time systems) [3] och i denna typ av system kan överskridningar av deadlines aldrig accepteras.

Däremot kan vissa realtidssystem tolerera att det inte finns garantier på att deadliner möts, och att de nu och då faktiskt missas. Dessa system klassificeras som mjuka realtidssystem (eng. soft real-time systems) [1]. I denna klass av realtidssystem nöjer man sig med att systemet oftast hinner exekvera inom sina utsatta deadliner. Dock vill man inte att det ska ske för ofta även om det inte utgör någon säkerhetsrisk, eftersom det innebär att systemet inte beter sig optimalt, exempelvis kan användarupplevelsen försämras. System som hanterar multimedia är ett klassiskt exempel på mjuka realtidssystem. Ifall uppspelningen av exempelvis en video inte sker tillräckligt snabbt blir videon hackig och användarupplevelsen dålig, men det äventyrar inte användarens hälsa. Deadliner i mjuka realtidssystem är således mer flexibla än i hårda realtidssystem, men det är ändå av intresse att kunna analysera systemet och även kunna påvisa att systemet oftast fungerar.

Abeni m.fl. [3] definierar en deadline i mjuka realtidssystem som den i genomsnitt

acceptabla responstiden i systemet istället för en övre gräns på responstid, såsom i hårda realtidssystem. Med responstid avses tiden mellan att ett nytt jobb skapas eller annars kan exekveras igen, och tidpunkten då jobbet faktiskt har exekverats till slut av processorn. Exempelvis kräver uppspelningen av en video med 60 bilder i sekunden att en ny bild ritas upp på skärmen med $\frac{1}{60}$sekunder, dvs. 16,7 ms mellanrum. Ifall systemet inte lyckas rita nästa bild i videon inom 16,7 ms blir uppspelningen hackig. Även om det är irriterande för tittaren är det inget större problem så länge videon inte kör fast alltför ofta. Denna tolkning gör det möjligt att tillämpa samma bevisföring på såväl hårda som mjuka realtidssystem. Däremot använder Leung m.fl. [7] en annan notation, där varje process har såväl en deadline som en önskad sluttid. Exempelvis kunde man önska att varje bild i videon ritas upp på skärmen, men lägga en strängare deadline att åtminstone varannan bild i videon ska ritas.

Gränserna på dessa klasser är luddiga, och ett datorsystem kan mycket väl innehålla processer av olika klasser. Det är uppenbarligt att bromsarna i en bil är ett hårt realtidssystem, medan radion utgör ett mjukt realtidssystem. Dessutom är det ofta möjligt att uppdatera mjukvaran i bilen. Processen som exekveras för att utföra uppdateringen har inga realtidskrav alls, eftersom den endast exekveras då bilen står stilla.

Ibland används även en tredje klass, tröga realtidssystem (eng. firm real-time systems), en klass som ligger mellan mjuka och hårda realtidssystem. I praktiken är hårda realtidssystem ett väldefinierat specialfall medan tröga, mjuka samt icke-realtidssystem är system med relativt mer flexibla deadliner utan någon exakt avgränsning. I denna avhandling tolkas tröga realtidssystem som en underklass av mjuka realtidssystem, och kommer således inte att behandlas närmare.

find ref

figur

Eftersom det börjar bli allt vanligare att ett givet realtidssystem omfattar flera processer med olika krav på responstid och mötning av deadliner, kan det vara mer exakt att tala om hårda och mjuka realtidsprocesser som Bertolotti m.fl. [1] gör i sin bok. Den definitionen kan utvidgas så att exempelvis termen hårt realtidssystem hänvisar till processerna (en eller flera) med hårda deadliner i ett större system bestående av flera olika klasser av realtidssystem.

4.3 Realtidsschedulerare

För att realtidssystemets alla processer alltid ska möta sina deadlines måste scheduleraren i operativsystemet anpassas ändamålsenligt. Med andra ord måste scheduleraren lyckas bestämma en ordning enligt vilken processerna i systemet kan exekveras på ett sådant sätt att så få processer missar sina deadliner som möjligt. Det är även viktigt att kunna matematiskt resonera om schedulerarens beteende, speciellt i värsta fall där möjligheten för att systemet misslyckas i scheduleringen är störst [3].

4.3.1 Målsättningen för realtidsschedulerare

I motsats till mer traditionella schedulerare betonar realtidsschedulerare processernas exekveringstid och deras deadliner mer än systemets totala effektivitet eller schedulerarens responstid[3]. Orsaken är att realtidsschedulerare inte nödvändigtvis kräver bra prestanda. Det enda som krävs är att det går att säkra sig om att systemet klarar av att exekvera alla processer inom sina respektive deadliner. Givetvis är optimering av realtidsschedulerarna även ett forskningsområde i sig.

5 Enkla processmodellen

De tre grundläggande egenskaperna för en realtidsprocess, nämligen dess deadline, värstafalls exekveringstid och periodicitet, kan användas som grund för en väldigt enkel processmodell. Processmodellen används för att analysera processernas beteende i systemet, och är väsentlig för senare bevis även om själva modellen är nästan självklar.

Dock avgränsar Bertolotti m.fl. [3] modellen ytterligare med ett flertal antaganden. För det första är antalet processer i systemet känt och konstant. Denna modell kan alltså inte hantera fallet där en ny process skapas eller tas bort ur systemet, utan en förändring i antalet processer måste hanteras som ett totalt nytt system av processer[2]. För det andra är alla processer periodiska, och deras perioder är oföränderliga. Processernas deadliner är hårda och sammanfaller med deras period. I praktiken måste alltså varje process exekveras färdigt innan den processen måste exekveras igen. Processerna kan endast exekveras en gång inom deras respektive period. Exekveringstiden för administrativa uppgifter i systemet, såsom kontextbyte från en process till en annan, är försumbara. Ytterligare är alla processer totalt oberoende av varandra. Inte ens enkla semaforer och synkroniseringsdirektiv mellan processerna får användas.

I verkligheten är det sällan som ett realtidssystem faller precis in i den enkla processmodellens ramverk, och det medger också Bertolotti m.fl. [3]. Modellen är ändå en bra startpunkt för att demonstrera hur matematisk bevisföring kring realtidsschedulerare kan utföras. Mer avancerade processmodeller kan användas för att behandla mer avancerade schedulerare, och sådana processmodeller behandlas i kapitel 8.

Figur

Alla figurer är placeholders

TABLE 11.1
Notation for real-time scheduling algorithms and analysis methods

Symbol	Meaning
τ_i	The i-th task
$\tau_{i,j}$	The j-th instance of the i-th task
T_i	The period of task τ_i
D_i	The relative deadline of task τ_i
C_i	The worst-case execution time of task τ_i
R_i	The worst-case response time of task τ_i
$r_{i,j}$	The release time of $\tau_{i,j}$
$f_{i,j}$	The response time of $\tau_{i,j}$
$d_{i,j}$	The absolute deadline of $\tau_{i,j}$

6 Exekveringsscheman

Utgående från den enkla processmodellen är det möjligt att skapa exekveringsscheman. Ett exekveringsschema är teoretiskt sett utdatan från en scheduleringsalgoritm. Bertolotti m.fl. [3] skiljer åt implementationen av scheduleringsalgoritmen i kod från det mer abstrakta systemet av regler enligt vilken algoritmen fungerar, och kallar det för scheduleringsmodellen. I denna avhandling används termen scheduleringsalgoritm såväl för implementationen som den abstrakta algoritmen istället, enligt samma stil som Abeni m.fl.[1].

Exekveringsschemat anger ordningen enligt vilken processerna i systemet bör exekveras, samt deras respektive deadliner, perioder och (värsta-falls) exekveringstider. De är också ett bra sätt att visuellt demonstrera hur en scheduleringsalgoritm kommer bete sig i en viss situation [3]. Figur X och Y är exempel på möjliga exekveringsscheman baserade på processerna som anges av tabell Z. Notera att exekveringen av process T_i i figur Y överskrider processens deadline. I detta fall har scheduleringen av systemet misslyckats, medan scheduleringen i figur X har lyckats eftersom alla processer hinner exekvera innan deras respektive deadliner.

figurer

TABLE 11.2
A simple task set to be executed by a cyclic executive

Task τ_i	Period T_i (ms)	Execution time C_i (ms)
τ_1	20	9
τ_2	40	8
τ_3	40	8
τ_4	80	2

Figur

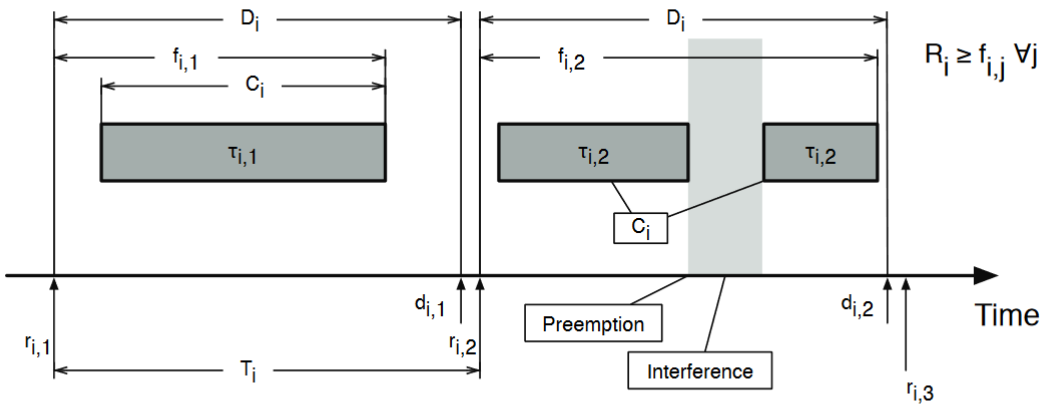

FIGURE 11.1
Notation for real-time scheduling algorithms and analysis methods.

6.1 Schedulerbarhet

Det framgår ur figur X att det existerar ett sådant exekveringsschema där alla processer i det givna systemet möter sina deadliner, dvs. att det givna systemet är schedulerbart. Eftersom analysen är baserad på värsta-falls exekveringstiden följer det att systemet alltid är schedulerbart för alla möjliga exekveringstider givet att de antaganden som användes i modelleringen av systemet håller.

6.2 Analys av användningsgrad

Vissa system av processer är omöjliga att schedulera med den beräkningskapacitet som processorn har. Tabell Å är ett sådant system, antaget atmt processorn som kör program- figur
met endast har en kärna. Orsaken varför systemet inte kan scheduleras är helt enkelt att de olika processerna kräver mer beräkningskapacitet än vad prosessorn har. Bertolotti m.fl. [3] presenterar ett bevis på att detta är fallet genom att beräkna en användningsgrad (eng. utilization factor) för prosessorn. Användningsgraden är ett tal (eller procent) som berättar hur stor andel av processorns totala beräkningskapacitet ett system av processer använder. Definitionen av användningsgraden är:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i}$$

och kan i ord beskrivas som summan av alla processers andel av processorns tota-

la beräkningskapacitet. En användningsgrad högre än 1 eller 100% kan aldrig uppnås, eftersom det skulle betyda att processorn exekverar mer än dess maximala beräkningskapacitet[3]. Trots beviset är det möjligt att ett system emellertid hinner exekvera ifall processerna i systemet råkar exekverar snabbare än deras värsta-falls exekveringstid, men i det allmänna fallet kan detta inte garanteras. figur

Det bör också noteras att ett bevis i samma stil gäller för mjuka realtidssystem. Ifall den genomsnittliga exekveringstiden för processerna används istället för värsta-falls exekveringstiden, är det möjligt att bevisa att systemet i det genomsnittliga fallet är schedulerbart. I en sådan analys uteslutes inte möjligheten att systemet emellertid inte uppnår sina deadlines. Å andra sidan kan ett sådant bevis vara tillräckligt, och att bevisa eller bygga ett system där det går att framföra ett starkare bevis på schedulerbarheten av systemet kan vara oöverkomligt svårt eller dyrt. ref?

7 Cyklisk schedulering

Ifall alla processer i ett system är periodiska och följer den enkla processmodellen, är det möjligt att definiera ett exekveringsschema i själva programkoden[3][2]. Oftast antas även att processers deadliner inte är längre än deras perioder, eftersom processering av data i det motsatta fallet skulle kräva något sorts databuffer[2]. Detta innebär att all schedulering i systemet definierats manuellt på förhand och scheduleraren helt enkelt byter från en process till en annan i en oändlig cykel. Denna metod av schedulering kallas cyklisk schedulering, cyklisk exekvering eller tidslinjesschedulering (eng. cyclic scheduling, cyclic executive, timeline scheduling). Cyklisk schedulering är en av de äldsta och enklaste formerna av realtidsschedulering[3][2]. Trots sin ålder används denna typ av schedulering fortfarande[3]. Cyklisk schedulering är speciellt användbart i enkla system där den enkla processmodellen är en nära beskrivning av verkligheten.

7.1 Stora och små cykler

Den centrala idén bakom algoritmen är att uppkomsten av processer i ett system med låsta perioder kommer repetera i cykler, som är lika långa som den största gemensamma faktorn av processernas perioder. Dessa cykler kallas stora cykler (eng. major cycles)[2]. I början av en stor cykel exekveras processerna i systemet enligt det inprogrammerade exekveringsschemat. Då en stor cykel tar slut, påbörjas en ny cykel. Det är naturligtvis möjligt att definiera flera olika exekveringsscheman i systemet och byta mellan dem, exempelvis för att göra det möjligt att byta från ett systemläge till ett annat. figur

Ytterligare brukar de stora cyklerna delas in i mindre, jämnstora delar som kallas små cykler eller ramar (eng. frames), men i princip är detta inte nödvändigt[2]. I början av figur

varje liten cykel vaknar processorn, exekverar de arbetsuppgifter i ordningen definierad i exekveringsschemat, och sover tills början av nästa liten cykel[2]. Alternativt kunde bakgrundsprocesser exekveras efter att den lilla cykeln exekverat färdigt, tills början av nästa lilla cykel.

De små cyklerna är nyttiga eftersom början av varje cykel påbörjas av en signal från en timer, vilket betyder att början av cykeln kan fungera som en synkroniseringspunkt[3][2]. Således garanteras att alla processer som bör exekveras endast körs en gång per ram oberoende av hur länge den förra ramen exekverade, givet att exekveringen av en process inte överskrider ramens gränser. Även det verkliga tidsintervallet mellan ramerna hålls konstant. Ifall den förra ramen fortfarande exekverar då nästa ram bör påbörjas, har scheduleringen av systemet misslyckats. Signalen från timern kan användas för att enkelt detektera när detta sker och försöka åtgärda situationen[3].

figur

8 Utvidgning av processmodellen

Utöver de antaganden som gjordes i den enkla processmodellen kan realtidsprocesser ha ett flertal andra egenskaper som scheduleraren kan använda. Ifall dessa är kända eller inte är fullt beroende av själva systemet, och en del egenskaper kan göra det svårare eller omöjligt att analysera systemets funktionerande[3]. Genom att beakta följande egenskaper är det möjligt att skapa en mer realistisk processmodell, vilket tillåter oss att resonera kring mer avancerade scheduleringsalgoritmer.

check ref

8.1 Aperiodiska processer

En del processer är icke-periodiska dvs. sporadiska till sin natur. Processer som exekveras i respons till någon händelse, t.ex. ett avbrott eller en extern signal, har ingen regelbunden periodicitet. Det följer att sådana processer omöjligen kan scheduleras offline. Det är också möjligt att processer delar på sig, terminerar, eller startar fler processer exempelvis som respons till användarens handlingar. System med oregelbundet varierande antal processer kan inte scheduleras på förhand, och scheduleras istället vid körtid. Alla scheduleringsalgoritmer som behandlas framöver är av denna typ.

8.2 Prioritet

8.3 Avbrott

8.4 Responstid

8.5 Kommunikation mellan processer

Inter-process kommunikation (IPC) används flitigt inom parallellprogrammering för att dela data mellan olika processer. IPC gör det möjligt att skapa komplicerande strukturer och beroendeförhållanden i program. Nackdelen med IPC är att det är just den ökade komplexiteten som gör systemen svårare att analysera. Detta reflekteras även i de många olika synkroniseringsproblemen som är vanliga inom parallellprogrammering. Ur schedule-ringsperspektiv är det speciellt farligt ifall en process med hög prioritet är beroende av en process med lägre prioritet, eftersom den lägre processen inte nödvändigtvis någonsin exekveras medan processen med hög prioritet endast väntar. Deadlocks kan naturligtvis inte tolereras, speciellt i kritiska system, och traditionella schedulerare försöker undvika deadlocks genom att upprätthålla en viss grad av rättvishet i systemet. I praktiken uppnås svenska det genom användningen av dynamiska prioriteter, dvs. att en process får en högre prioritet ju längre den väntat på att få exekvera. Så kommer prioriteten för processer med låg prioritet emellertid stiga över prioriteten av andra processer, vilket löser deadlocken.

Realtidsschedulerare erbjuder i regel inte en sådan möjlighet, och det är upp till utvecklaren att se till att deadlocks inte kan ske. Även om de klassiska problemen med ref? parallellprogrammering löses, kan det vara svårt att definiera en exakt värsta-falls körtid för processer som är beroende av indata från andra processer. Ifall det inte är möjligt att omstrukturera systemet så att realtidsprocesserna är oberoende från varandra, måste uppskattningar av körtiderna användas vid analys av systemet.

8.6 Overhead

8.7 Processorer med flera kärnor

8.8 Uppskattningar och heuristik

9 Prioritetsschedulering

10 Rate Monotonic

10.1 Bevis

11 Earliest Deadline First

11.1 Bevis

12 DMPO-algoritmen

13 Schedulering i processorer med flera kärnor

14 Sammanfattning

[6] [1] [8]

Källförteckning

[1] L. Abeni och G. Buttazzo. ”Integrating multimedia applications in hard real-time systems”. I: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*. 1998, s. 4–13. DOI: 10.1109/REAL.1998.739726.

[2] T.P. Baker och A. Shaw. ”The cyclic executive model and Ada”. I: *Proceedings. Real-Time Systems Symposium*. 1988, s. 120–129. DOI: 10.1109/REAL.1988.51108.

[3] Ivan Cibrario Bertolotti och Gabriele Manduchi. *Real-Time Embedded Systems: Open-Source Operating Systems Perspective*. Taylor & Francis Group, 2012. ISBN: 9781439841617.

[4] Georg von der Brüggen m. fl. ”Efficiently Approximating the Worst-Case Deadline Failure Probability under EDF”. I: *2021 IEEE Real-Time Systems Symposium (RTSS)*. 2021, s. 214–226. DOI: 10.1109/RTSS52674.2021.00029.

[5] E.W. Dijkstra. ”Co-operating sequential processes”. English. I: *Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys*. United States: Academic Press Inc., 1968, s. 43–112. ISBN: 0-12-279750-7.

[6] Wei-Cong Fan m. fl. ”Comparison of Interactivity Performance of Linux CFS and Windows 10 CPU Schedulers”. I: *2020 International Conference on Green and Human Information Technology (ICGHIT)*. 2020, s. 31–34. DOI: 10.1109/ICGHIT49656.2020.00014.

[7] Joseph Y-T Leung. *Handbook of Scheduling. Algorithms, Models, and Performance Analysis*. Chapman och Hall/CRC, 2004. ISBN: 9780429205644.

[8] Jianpeng Li m. fl. ”Task Scheduling Algorithm for Heterogeneous Real-time Systems Based on Deadline Constraints”. I: *2019 IEEE 9th International Conference on Electronics Information and Emergency Communication (ICEIEC)*. 2019, s. 113–116. DOI: 10.1109/ICEIEC.2019.8784641.