Universitatea Politehnica București

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Temă de curs Inginerie Software

Embedded Software

“Aplicații în timp real”

 Badea Mihai-Sorin

 Guliman George

Grupa 442A

București 2015

Cuprins

1. **Introducere - Guliman George**
2. **Proiectarea produselor software în timp real - Guliman George**
3. **Modele arhitecturale pentru software embedded – Badea Mihai-Sorin**
4. **Succesiunea temporală a proceselor (timing analysis and scheduling) – Badea Mihai-Sorin**
5. **Bibliografie**
6. **Introducere**
	1. Procesare în timp real

Calculul în timp real sau calcului reactiv este studiul sistemelor hardware și software care sunt ținta constrângerilor de timp cum ar fi termenele limită de la producerea unui eveniment la răspunsul sistemului.

Aplicațiile în timp real trebuie să garanteze răspunsuri care să se încadreze în constrângerile de timp impuse de sistem. Aceste răspunsuri trebuie să fie de ordinul milisecundelor/microsecundelor. Aceste cerințe variază în funcție de sistemul respectiv și scopul acestuia.Un sistem fără facilități de procesare în timp real nu poate garanta răspunsul într-un interval de timp presetat.

Un sistem în timp real are funcția de a controla un mediu prin recepționarea de date, procesarea acestora si returnarea unor rezultate într-un timp sufficient de bun pentru a putea controla mediul la un moment dat. [3]

Un software în timp real poate utiliza programarea sincronă, sisteme de operare în timp real sau rețele în timp real fiecare din acestea furnizând un schelet essential pe care să se construiască o aplicație în timp real.

Sistemele în timp real sunt de regulă utilizate pentru aplicațiile foarte importante în care întârzierile nu pot fi acceptate. Putem lua ca exemplu sistemele ABS de pe autoturisme. Aici, constângerea în timp este data de timpul în care frânele trebuie activate pentru a împiedica blocarea roților.

Un sistem poate fi catalogat ca sistem în timp real dacă corectitudinea unei operații nu depinde numai de corectitudinea logică a acesteia ci și de timpul în care aceasta a fost efectuată. Sistemele în timp real, la fel ca și constrângerile de timp ale acestora sunt clasificate în funcție de consecințele apărute în cazul în care o asemenea constrângere nu este respectată:

* Hard – ratarea unui termen limită înseamnă un eșesc total al sistemului
* Firm – ratarea termenelor limită ocazional este tolerabilă însă afectează perfomanța sistemului. Utilitatea unui rezultat este nulă după depașirea termenului limită
* Soft – Utilitatea unui rezultat degradează după depașirea termenului limită[3]

Scopul unui sistem în timp real de tip “Hard” este să asigure că toate termenele limită sunt respectate însă pentru un sistem de tip “Soft” este să asigure un anumit subset de termene limită specifice aplicației.

Sistemele de tip hard sunt utilizate când este foarte important ca răspunsul la un eveniment să vină după un termen limită strict. Asemenea asigurări sunt necesare în cazul sistemelor în care dacă nu avem un răspuns în intervalul stabilit de timp se provoacă pierderi omenești sau bunuri.

În contextual sistemelor multi-tasking politica de programare este făcută în funcție de prioritatea unui anumit process.

 Sistemele de tip soft sunt de regulă utilizate pentru a rezolva problemele de acces concurrent. Un exemplu poate fi reprezentat de un software ce tine la zi baza de date a zborurilor pentru companniile aeriene.[3]

 Procesarea în timp real este foarte des confundată cu procesarea de performanță înaltă însă aceasta nu este o clasificare precisă. De exemplu, un supercomputer care execută simulări scințifice poate oferi performanțe impresionante însă acesta nu efectuează calcule în timp real. În cazul sistemelor de acest gen întârzierile sunt premise acestea putând să își ducă la capăt sarcina în cele din urmă.

 Cea mai importantă cerință într-un sistem de timp real rămâne previzibilitatea ci nu neapărat performanța.

* 1. Embedded systems

Un sistem embedded este un computer ce are o funcție dedicată într-un sistem mechanic sau electric mai mare care are adesea si contrangeri de procesare în timp real. Acesta este încorporat într-un dispozitiv care cel mai adesea conține componente mecanice și electrice. Sistemele embedded controlează multe dintre dispozitivele uzuale din ziua de astăzi.[4]

 Proprietățile tipice ale acestor sisteme în comparație cu calculatoarele universal sunt: consum redus de putere, dimensiuni mici, condiții de funcționare dificile și costul per unitate redus. Acestea vin cu prețul resurselor limitate de procesare și faptul că sunt mult mai greu de programat si de interacționat cu ele.[4]

 Sistemle embedded moderne se bazează cel mai adesea pe microcontraollere dar se mai întâlnesc și microprocesoare obișnuite mai ales în cazul sistemelor mai complexe. O clasă comună de proceasoare dedicate este reprezentată de procesoarele de semnal (DSP).

 Sistemle embedded sunt proiectate pentru a îndeplini o sarcină specifică spre deosebire de calculatoarele de uz general care sunt proiectate pentru indeplinirea a unor sarcini mai diverse și complexe. Majoritatea sistemelor embedded au și capabilități de procesare în timp real cu constrăngeri care trebuie îndeplinite pentru a își îndeplini funcțiile.

 Programele scrise pentru astfel de sisteme se numesc “firmware” și sunt stocate într-o memorie de tip read-only sau memorie flash. Acestea rulează cu resurse hardware limitate.

 În măsură ce complexitatea acestor sisteme crește, sunt necesare unelte de programare de nivel din ce în ce mai înalt iar au început ca și sistemle de operare să migreze pe astfel de medii acolo unde are rost. Putem lua ca exemplu telefoanele mobile, PDA-urile și alte astfel de calculatoare dedicate clienților care necesită componente software semificative ca achiziționat de o persona, alta decât producătorul componentelor electronice. În aceste sisteme un mediu de dezvoltare open source precum Linux, NetBSD, sau Embedded Java este necesar pentru ca software-ul să fie vândut pe piațele de deschidere.[4]

* 1. Embedded software

Softwareul embedded este tema principal a acestei lucrări capitolele precedente fiind inițiere pentru a ne ajuta să înțelegem mai bine pentru ce este necesare acest tip de dezvoltare software și care sunt aplicațiile acesteia.

Aplicațiile de tip embedded software sunt programe de mașini de calcul scrise pentru a controla dispositive care nu sunt în mod tipic privite ca fiind calculatoare. Aceste aplicații sunt în mod tipic specializate pentru componentele hardware particulare pe care acestea rulează și au constrângeri de timp și de memorie. Acest termen este de cele mai multe ori utilizat inter-schimbabil cu conceptual de firmware. [4]

O caracteristică foarte des întâlnită este că nu toate funcțiile găsite în acest timp de software sunt controlate sau inițiate de o interfață umană ci de interfețe mașină.

Producătorii utilizează aplicații de acest tip în componentele electronice utilizate pe mașini, telefoane, modemuri, sisteme de securitate, etc. Acest software poate fi foarte simplu cum ar fi controlarea unor led-uri de care sunt necesare un microcontroller de 8 biți și câțiva kilobiți de memories sau pot fi foarte complexe și sunt utilizate în avioane, rachete, etc..

Timpul de răspuns în timp real este cea mai importantă diferență dintre sistemele embedded și alte sisteme software, al căror scop principal este procesarea de date. Pentru un sistem care nu este real-time corectitudinea acestuia poate fi definită precizând cum inputurile sistemului corespund cu outputurile acestuia. Pentru a răspunde unei intrări, o ieșire corespunzătoare ar trebui să fie generate de sistem, și, cel mai adesea, un set de date ar trebui sa fie stocate. De exemplu, dacă alegem o comandă de creare într-un sistem ce menține datele unor pacienți , atunci răspunsul correct al sistemului ar fi să creeze o intrare a unui nou pacient într-o bază de date și să confirme că acest lucru a fost făcut. Dacă toate acestea se încadrează în niște limite rezonabile, nu contează cât de mult durează.[4]

Spre deosebire de cazul anterior, într-un sistem în timp real, corectitudinea depinde și de rezultatul unei intrări dar și de timpul necesar pentru generarea acelui răspuns. Dacă sistemului îi ia prea mult să genereze un rezultat atunci acel răspuns poate ajunge să fie inutil. De exemplu, dacă un software embedded care controlează sistemul de frânare al unei mașini este prea încet în a oferi un răspuns, un accident poate avea loc deoarece ar fi imposibil de oprit mașina în timp util.

Așadar, timpul este factor foarte important în definirea unui sistem software în timp real.

De la afirmațiile de mai sus putem adăuga următoarea definiție:

Un sistem software în timp real este un sistem ale cărui operații corecte depend și de corectitudinea rezultatului produs de sistem dar și de timpul la care aceste rezultate au fost produse. Un sistem în timp real “soft” este un sistem ale cărui operații sunt degradate dacă răspunsul nu este furnizat în constrângerile de timp definite pentru sistem. Daca același lucru se întâmpla pentru un sistem în timp real “hard” acest lucru este considerat un eșec al sistemului.[9]

Sistemele embedded pot fi considerat sisteme reactive: acest lucru înseamnă că sistemele trebuie să reacționeze la evenimente la viteza mediului în care acestea sunt utilizate. Timpii de răspuns sunt adesea guvernate de legile fizicii în pofida unor cazuri alese conventional de intervenția umană.[9]

1. **Proiectarea produselor software în timp real**

În procesul de proiectare a unui sistem embedded, dezvoltatorii softwareului trebuie sa ia în considerare proiectul și performanțele sistemului din punct de vedere hardware. O parte a procesului de proiectare implică și luare deciziei cu privire la ce caracteristici vor fi implementate hardware ș ice caracteristici vor fi implementate software. Multe din sistemele în timp real găsite în produsele de consum de pe piața cum ar fi sistemele găsite în telefoanele mobile, costul și consumul de putere al parților electronice sunt esențiale. Din acest motiv, în procesul de proiectare se poate alege utilizarea unor tipuri specific de micro-procesoare care suportă sisteme embedded.

Constrângerile pe care le întâmpină o echipă de proiectare reprezintă în cele din urma cerințele unui sistem. Acestea nu sunt însă prederminate și pot fi modificate pe durata proiectarii. Majoritatea cerințelor definesc o valoare minimă și maxima ci una fixată. Așadar aceste cerințe pot fi considerate limitele spațiului de proiectare. În spațiul de proiectare toate soluțiile fezabile pot fi luate în considerare. Din toate aceste soluții proiectantul este interesat de cea mai optimă soluție. După cele spuse deducem că este nevoie de untilizarea a cel puțin unei metode de optimizare. Cele mai commune astfel de metode sunt:

* Optimizare totala
* Utilizarea principiului Pareto: 20% din efortul depus pentru prima metodă atinge 80% din scopul acesteia
* Best effort
* Programare liniară
* Optimizare multi-obiectiv
* Inginerie bazată pe modellare

Motivele pentru care optimizarea unui proiect este complicată sunt următoarele:

* O schimbare într-unul din criteriile luate în cosiderare pentru procesul de proiectare are un impact ăîn toate celelalte criteria. De aici rezultă că optimizarea trebuie făcută luând în considerare toate criteriile simultan ci nu pentru fiecare criteriu în parte.
* Criterrile de proiectare nu se pot reprezenta întotdeauna ca valori numerice
* Incertitudini legate de cerințe care crează incertitudini în spațiul/mediul de proiectare[1]

Într-un caz ideal, toate cerințele sunt convertibile în forme de reprezentare cantitative pe care proiectantul le poate optimiza. Însă, în cele mai multe cazuri reale, unele dintre cerințe nu se pot reprezenta prin valori numerice iar prin urmare optimizarea în aceste cazuri nu este atât de ușoară precum, de exemplu, minimizarea sau maximizarea unei valori. Costul de procesare a optimizării nu este în cele din urmă singurul factor de luat în considerare. Cuantizarea criteriilor de proiectare poate fi foarte importantă și uneori de incertitudine foarte ridicată însă nu poate fi evitată.

Robustețea este un aspect important în proiectarea unui sistem. Aceasta diferă de cele mai multe criterii de proiectare din cauza dificultății de a o cuantifica sau a o măsura. Un sistem este considerat a fi robust atunci când este în măsură să facă față la incertitudini și variații ale mediului operațional, cu pagube minime și fără a își pierdere funcționalitatea. Optimizarea și robustețea pot fi vazute ca fiind complementul fiecaruia (acestea sunt dependete între ele). Optimizarea asigură că, în mediul operațional sistemul funcționează optim (de exemplu, costul minim), robustețe trebuie să se asigure că, în cazul în care apare un motiv necunoscut, de exemplu mediul și schimbă sistemul, schimbările respective nu vor cauza deteriorarea și pierderea funcționalității.[1]

În concluzie, putem afirma că: proiectantul întotdeauna încearcă să găsească o soluție optimă din intervalul a mai multor soluții fezabile. Acest interval poartă numele de spațiu de proiectare. Înainte ca tehnicile de optimizare să poată fi utilizate, fiecare criteriu de proiectare trebuie să fie cuantizat însă acest lucru poate fi dificil uneori. O bună proiectare implică un balans bine gândit între optimizare si robustețe.

În continuare vom aborda mai specific, subiectul proiectării aplicațiilor în timp real.

Ținând cont de faptul că sistemele embedded sunt sisteme reactive care reacționează la evenimente care se petrec în mediul de utilizare, abordarea cea mai generală în proiectarea aplicațiilor în timp real este bazată pe un model stimuli-răspuns. Un stimul este reprezentat de un eveniment ce a avut loc în mediul înconjurator sistemului care cauzează ca sistemul să reacționeze într-un anumit mod. Un răspuns este de regulă un semnal sau un mesaj care este trimis de software în mediul corespunzător lui.[9]

Comportamentul unui sistem sistem real-time poate fi definit determinând stimuli recepționați de sistem, raspunsurile associate și timpul în care un răspuns trebuie produs.

Stimuli se pot împărți în 2 clase:

1. Stimuli periodici: au loc la interval determinate de timp. Exemplu: sistemul poate acționa un sezor la fiecare 10 milisecunde și depinzând de răspuns poate efectua o anumită acțiune
2. Stimuli aperiodici: aceștia au în mod neregulat și sunt imprevizibili și sunt de obicei semnalizați de mecanismul de control al întreruperilor. De exemplu: o întrerupere ce indică un transfer I/O care a fost completat.[9]

Table 1:Stimuli și răspunsuri pentru un sistem anti-furt

|  |  |
| --- | --- |
| Stimul | Răspuns |
| Un singur sensor este pozitiv | Inițiază o alarmă; Aprinde luminile în jurul senzorului pozitiv |
| Doi sau mai mulți senzori pozitivi | Inițiază alarma; aprinde luminile în jurul senzorilor pozitivi; sună la poliție |
| Scădere tensiune între 10%-20% | Schimbă pe baterie; efectuează teste la sursa de tensiune |
| Scădere tensiune cu mai mult de 20% | Schimbă pe baterie; inițiază alarmă; sună la politie; inițiază test pe sursa de tensiune |
| Sursa de tensiune defectă | Sună tehnician |
| Senzor defect | Sună tehnician |
| Buton de panică pozitiv | Inițiază alarmă; |
| Stergere alarme | Închide toate alarmele active; stinge toate luminile care au fost aprinse |

 Stimulii vin de la senzorii și răspunsurile sunt trimise la un actuator. De cele mai multe ori este bine să avem procese diferite pentru fiecare tip de senzor și actuator. Acești actoatori controlează exhimpamentele care apoi fac schimbări în mediul sistemului. Actuatorii pot genera la rândul lor stimuli. Stimulii generate de actuatori indică de regulă faptul că o problemă a intervenit.



URL: http://www.adi.com/technology/tech-apps/what-is-real-time-technology/

Figure 1:Model general a unui sistem real-time

Pentru fiecare tip de senzor, poate exista un process pentru controlul acestuia care se ocupă de colecționarea de date de la senzori. Aceste procese calculează răspunsurile necesare pentru stimuli primiți de sistem. Procesele de control ale actuatorilor sunt associate cu fiecare din actuatori și controlează acțiunea acestuia. Acest model permite datelor să fie colecționate rapid de la senzori și permite ca procesarea și răspunsul asociat unui actuator să fie effectuate mai târziu.[9]

Figure 2:Procese de sensor și actuator[9]

Un sistem în timp real trebuie sa răspună la stimuli care au loc în momente diferite de timp. Din acest motiv trebuie ca arhitectura de sistem să fie organizată în așa fel încât de îndată ce un stimul este recepționat, operația de control este transferată manipulatorul corespunzător. Acest lucru este imposibil în programele secvențiale. În consecință, programele real-time sunt în mod normal proiectate ca un set de procese concurente care comunică între ele. Pentru a suporta astfel de procese, platform pe care rulează aplcația trebuie să include un sistem de operare în timp real.

Deși nu există un process standard de proiectare putem spune ca următoarele activități pot fi incluse în procesul de proiectare a unei aplicații în timp real:

1. **Selectarea unei platforme**. Acest lucru include și partea hardware dar și alegerea unui sistem de operare în timp real.
2. **Identificarea stimulilor și răspunsurile aferente acestora.**
3. **Analiza în timp**. Pentru fiecare stimul si răspunsul asociat acestuia identificăm constrângerile de timp care se aplică atât pe stimul cât și pe timpul de procesare al unui răspuns.
4. **Proiectarea proceselor**. Se face agregarea procesării stimulului și a răspunsului acestuia într-un număr de procese concurente.
5. **Proiectarea algoritmului**. Pentru fiecare stimul și răspuns trebuie proiectați algoritmi care efectuează procesările dorite.
6. **Proiectarea datelor**. Specificarea informațiilor ce trebuie schimbate între procese și evenimentele care coordonează schimbul de informații.
7. **Process scheduling**. Proiectarea unui sistem de scheduling care să asigure că procesele sunt pornite în timp util astfel încât să își ideplinească termenele limită.[9]

Când proiectăm schimbul de date între procese trebuie să ținem cont de faptul că aceste procese pot rula la viteze diferite. Un proces produce informații iar alt process le poate consuma. Vom numi aceste procese producător și consummator. Dacă producătorul rulează mai rapid decât consumatorul, informațiile noi pot supra-scrie o informative ce a fost abia a fost citită înainte ca procesul consummator să apuce să citească prima informție scrisă. Dacă procesul cumator rulează mai rapid decât cel producător, aceeași informative poate fi citită de două ori.[9]

Pentru a rezolva această problemă schimbul de informații trebuie implementat folosind un buffer comun și accesul la acel buffer trebuie controlat.

URL: http://www.camera-sdk.com/p\_54-how-to-implement-circular-buffer-video-recording-onvif.html

Figure 3: Buffer comun

 În figura de mai sus se observă ca operațiile de citire din buffer se efectuează din capul buffer-ului iar cele de scriere se efectuează în coada acestuia deci, procesul consummator o să citească întotdeauna din capul buffer-ului iar cel prducător din coada acestuia.

 Mai trebuie să ne asigurăm ca procesul consummator și cel producător nu încearcă să acceseze același element în acelasi moment de timp, adică în cazul în care Capul=Coada. Pentru a realiza acest lucru implementăm operațiile de insert și extract pentru accesul la buffer. Operația de insert este apelată de procesul producător iar cea de extract de extract de cel consummator. Pentru a ne asigura că aceste două opearații sunt sincronizate putem utiliza semafoare astfel încât acestea să nu acceseze aceeași regiune în același timp.[9]

 Odată ce am ales platform de execuție pentru sistem, am proiectat o arhitectură de procese și ne-am hotărât asupra unei politici de gestionare a proceselor trebuie să verificăm dacă sistemul își îndeplinește constrângerile de timp. Putem face acest lucru printr-o analiză statică a sistemului sau putem face analiza în timp a sistemului. Dacă indetificăm o problemă acest lucru înseamnă că trebuie să ne gandim la alternative de implementarea cum ar fi implementarea unei anumite funcții folosind componente hardware și așa mai departe.

 Evenimentele la care un sistem în timp real trebuie să acționeze poate să îl facă pe acesta să treacă dintr-o stare în alta. Pentru a descrie mai bine astfel de cazuri se poate folosi limbajul UML pentru definirea unei diagrame de stare.[9]

URL: https://www.google.ro/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0CAYQjB0&url=http%3A%2F%2Fseg.nju.edu.cn%2FBASIS010%2Fatm\_full.htm&ei=POTYVIXzC5PXao33gXg&bvm=bv.85464276,d.d2s&psig=AFQjCNF--F2cyc7iWPXw5\_1dUcvXKrhXCg&ust=14235866194219

Figure 4:Exemplu diagramă de stare

 Pentru a scrie efectiv programul în timp real trebuie să utilizăm un limbaj de programare în timp real. Aceste limbaje de programare trebuie să include facilități pentru accesul la hardwareul sistemului și ar trebui să ne permită să prezicem durata de timp ale operațiilor. Sistemle în timp real de tip hard sunt cel mai adesea programate în limbaj assembley pentru ca constrângerile de timp să poată fi îndeplinite. Limbajele de nivel mai înalt precum C care ne permit să generăm cod efficient sunt de asemenea foarte răspândite.

 Avantajul utilizării unui limbaj precum C constă în faptul că ne permite să dezvoltăm un program foarte efficient. Dezavantajele sunt reprezentate de faptul că acest tip de limbaje nu includ funcții/metode care să suporte accesul și managementul la resurse commune. Acest lucru este realizat de regulă prin apeluri ale unor funcții speciale care sunt înglobate în sistemul de operare în timp real utilizat. Aceste funcții sunt de regulă semafoare însă acestea nu pot fi verificate de către compilator așadar erorile de programare pot fi făcute foarte ușor. Programele sunt și mai greu de înțeles deoarece limbajul nu include caracteristici de timp real.[9]

 Deoarece sistemele în timp real au constrângeri de timp foarte stricte utilizarea unui limbaj obiect-orientat poate fi greu de implementat. Dezvoltarea obiect-orientată implică ascunderea datelor si accesarea atributelor prin operații definite în obiect. Asta înseamnă că este nevoie de rularea de cod suplimentar pentru medierea accesului la attribute și apelarea metodelor.

 Există însă o versiune de java dedicate sistemelor de acest tip. Acest limbaj se numeste Real-Time Java și a fost dezvoltat de Dibble. Acest limbaj include un mecanism de gestiune a thread-urilor modicat care permite specificarea unui thread care să nu fie interrupt de mecanismul de garbage collection. Gestiunea evenimentelor în mod asincron și specificarea timpilor de execuție a fost de asemenea adăugat. Totuși trebuie precizat că acest limbaj de programare este cel mai adesea utilizat pe sistemele care au resurse considerabile de memorie și procesare. Pentru sistemele cu resurse mail imitate se folosește cel adesea limbajul C.[9]

**3. Modele arhitecturale pentru software embedded**

Modelele arhitecturale reprezintă o îngolbare a cunoștiințelor referitoare la modurile de organizare ale arhitecturii sistemelor, la situațiile în care aceste arhitecturi ar trebui să fie folosite, și când nu ar trebui.

Sistemele embedded sunt făcute în mod frecvent pentru a controla, monitoriza sau ajuta alte sisteme (echipamente, mașinării). Spre exemplu, un sistem de control embedded este un produs software care poate fi folosit la controloarea mașinăriilor de dimensiuni mari (cum ar fi unele folosite in agricultură sau minerit). Sistemele de acest tip se află în strânsă legătura cu mediul înconjurător, fapt ce a dus la apariția unor necesități pentru astfel de sisteme embedded, precum: să fie în timp real, să fie sisteme distribuite, etc.

Cu cât aceste sisteme au crescut din punct de vedere al dimensiunilor, nevoia de a folosi anumite modele arhitecturale pentru software a devenit crucială în ceea ce privește calitatea produselor.

 Diferențele între software-ul embedded și cel interactiv duc la existența unor modele arhitecturale cu totul diferite pentre cele două tipuri de software. Modelele specifice sistemelor embedded sunt proces-orientate, spre deosebire de alte modele cum ar fi cel obiect-orientat sau cel orientat spre componente. [8],[9]

 În continuare vor fi abordate următoarele modele arhitecturale pentru aplicațiile în timp real:

1. Observare și Reacționare
2. Controlul Mediului
3. Pipeline-ul Proceselor
4. Alocarea si Programarea Fixă a Proceselor
5. Programarea Statică cu Ceas Extern
6. Alocarea Fixă a Memoriei

Aceste modele pot fi combinate între ele, iar într-un singur sistem nu este garantată prezența unui singur model. De exemplu, atunci când este folosit modelul de tip Controlul mediului, actuatorele sunt de obicei monitorizate folosind un model de tip Observare și Reacționare.

1. **Observare și Reacționare**

Acest model arhitectural software este folosit foarte des în sistemele de monitorizare, o clasă important a sistemelor embedded în timp real. Aceste sisteme adună informații despre mediul înconjurător folosind o serie de senzori și afișează informații despre starea mediului. Afișarea poate fi realizată în mai multe moduri: pe un ecran inclus în sistem, pe display-uri separate special pentru această acțiune sau pe un display aflat la distanță mare de sistemul fizic. Dacă are loc vreun eveniment special (sau o stare neobișnuită a senzorului este detectată), sistemul de monitorizare alege o modalitate de reacționare, de obicei prin pornirea unei alarme care să atragă atenția utilizatorului (operatorului) urmată, eventual, de oprirea sistemului.

Un sistem de monitorizare poate folosi modelul de tip Observare și reacționare în mai multe instanțe, pentru fiecare tip de senzor întâlnit. [9]

1. **Controlul Mediului**

Software-ul embedded este folosit cu precădere în sistemele de control. În aceste sisteme, software-ul are rolul de a controla diverse echipamente pe baza datelor primite de la senzori, cu scopul de a modifica condițiile în care se află sistemul. Aceste sisteme folosesc modelul arhitectural Controlul Mediului. Acest model este considerat ca fiind unul foarte general.

Un exemplu foarte important și foarte des întâlnit al acestui model arhitectural este folosirea lui în sistemele ABS (Anti-lock Braking System). Aceste sisteme sunt o parte crucială în ceea ce privește nivelul de siguranță al unui automobil. ABS permite anumitor roți să nu fie blocate de frână, evitând astfel derapaje necontrolate.[9]



Arhitectura unui sistem de control pentru un sistem de frânare anti-derapaj

(lustrație din [9])

1. **Pipeline-ul Proceselor**

În foarte multe cazuri, sistemele în timp real colectează date din mediu și îi schimbă forma de reprezentare originală într-una digitală care poate fi analizată și procesată mai bine de către sistem. După prima transformare, datele (acum digitae) poate fi convertit înapoi într-un format analog și retransmise în mediu.

În cadrul acestui tip de sisteme, procesarea datelor trebuie să fie făcută foarte rapid, pentru a preveni probleme precum pierderea datelor care sosec în timpul procesării. Modelul arhitectural de tip Pipeline-ul Proceselor face posibilă procesarea foarte rapidă a datelor de intrare prin separarea etapelor de procesare într-o serie de transformări independente, fiecare transformare având drept corespondent un proces unic, independent.

Acest model arhitectural software devine cu atât mai eficient cu cât și arhitectura hardware este adaptată acestui concept, prin folosirea procesoarelor multi-nucleu sau a mai multe procesoare. În cazul unui sistem multi-nucleu, fiecare proces (care corespunde unei transformări) este asociat unui nucleu. Astfel, pașii procesării sunt realizați în paralel.

În cazul folosirii pipeline-ului, pentru a face legătura între diversele procese, se folosește adesea o serie de memorii tampon (buffers) sincronizate, pentru a permite proceselor să ruleze la viteze diferite. [9]

1. **Alocarea si Programarea Fixă a Proceselor**

Acest model poate fi folosit atunci când nu știm nivelele de prioritate ale proceselor dar anumite procese trebuie repetate frecvent, iar altele au timpi de execuție mai mari. În acest caz, trebuie cumva create procese cu un comportament temporal previzibil.

În acest caz nu se cunosc timpii de inițializare așa că toate resursele trebuie alocate la pornirea sistemului. Proiectanții împart sistemul în blocuri executabile, fiecare bloc având un timp de execuție separat. Se calculează timpii de execuție în cazul cel mai nefavorabil, timpi după care se face programarea în timp a proceselor. [8]

1. **Programarea Statică cu Ceas Extern**

Acest model reprezintă o variantă îmbunătățită a modelului anterior. El asigură sincronizarea între diversele noduri de procesare care rulează procese în timp real. În acest caz, ca și înainte, este nevoie de programarea fixă în timp pe baza timpilor de procesare din cazurile cele mai nefavorabile. După ce sosește un semnal de sincronizare de la un ceas extern, un nou ciclu de programări este inițializat. După execuția tuturor proceselor, sistemul așteaptă următorul semnal de sincronizare. Dacă în intervalul cuprins între două semnale de sincronizare sistemul nu reușește să execute toate procesele, este dat un semnal de eroare.

Rezultatul va fi un sistem al cărui componente vor fi mereu sincronizate. Astfel toate termenele vor fi mereu respectate.[8]

1. **Alocarea Fixă a Memoriei**

Modelul acesta este folosit în cazul unui sistem cu o cantitate restrictivă de memorie. De asemenea, acest model este folosit doar dacă se cunoaște cantitatea maximă de memorie necesară pentru datele stocate.

Folosirea acestui model arhitectural de software duce la înlăturarea oricărui risc pus de alocarea defectuoasă a memoriei. Altfel ar fi necesară implementarea unor măsuri de siguranță pentru cazul alocării fără succes a memoriei. Aceste măsuri de siguranță ocupă spațiu și duc la o putere de procesare necesară mai mare. Alternativa unei alocări dinamice a memoriei este scoasă din discuție, datorită cantității mici de memorie. Sistemul rezultat este unul în care nu mai este necesară alocarea memoriei după inițializarea programului.[8]

**4. Succesiunea temporală a proceselor (timing analysis and scheduling)**

Sistemele care își desfășoară activitatea în timp real nu sunt considerate ca fiind bune doar analizând ieșirile, ci și momentele de timp la care au apărut ieșirile. Acest fapt duce la necesitatea unei analize în timp, care determină ordinea în care trebuie executate procesele în timp pentru asigurarea procesării intrărilor și producerii răspunsului sistemului la timp .

 Problema realizării acestei analize este dată de faptul că sistemele embedded care lucrează în timp real depind de evenimente din mediul înconjurător, evenimente care nu pot fi prezise (evenimente asincrone). Un exemplu de eveniment care nu poate fi prezis este pana de curent. În cazul în care are loc o pană de curent, sistemul trebuie să oprească toate subsistemele într-un mod controlat, într-un interval de timp foarte scurt. Totuși, progresele în domeniul vitezelor de procesoare ale echipamentelor moderne a reușit să permită o proiectare a sistemului ținând cont doar de evenimente periodice. În cazul precedent (pană de curent), ar putea fi implementat un proces care verifică periodic (cu o perioadă foarte mică) starea alimentării sistmeului. Dacă această verificare este executată suficient de frecvent, este timp pentru rularea rutinei de închidere controlată a subsistemelor.

 În cadrul analizei sincronizării, trebuie luate în considerare următorii factori-cheie:

1. **Termenele limită** – reprezintă timpul în care un stimul trebuie să fie procesat, iar sistemul să furnizeze un răspuns. Într-un sistem în timp real „hard”, ratarea unui deadline duce automat la erori de sistem.
2. **Frecvența** – reprezintă numărul de repetări într-o singură secundă a unui proces pentru a avea certitudinea atât a identificării unor evenimente (precum pana de curent) cât și a îndeplinirii termenelor limită de fiecare dată.
3. **Timpul de execuție** – reprezintă timpul necesar procesării unui stimul și furnizării unui răspuns. Când se discută despre timpul de execuție, se discută în mod frecvent despre timpul mediu de execuție și timpul de execuție în cel mai nefavorabil caz **(Worst-Case Execution Time – WCET**). Timpii de execuție nu sunt mereu aceeași din motive precum existența cazurilor diferite determinate de intrări diferite sau timpii de așteptare cauzați de așteptarea răspunsurilor altor procese, etc. Pentru asigurarea îndeplinirii termenelor limită, se poate lua în considerare doar timpul de execuție în cazul cel mai nefavorabil.

Punctul de start pentru orice analiză temporală în ceea ce privește procesele sistemului este reprezentat de alegerea tuturor termenelor limită. Continuând cu exemplul anterior, dacă are loc o pană de curent, termenul limită pentru oprirea controlată a subsistemelor trebuie să fie mai mic decât timpul în care s-ar ajunge la o tensiune suficient de mică pentru a cauza defecțiuni sistemului. În practică, pentru acest caz s-ar alege un termen limită considerabil mai mic pentru a exista o oarecare marjă de siguranță. [9]

După cum a fost precizat anterior, în cadrul analizei, se poate analiza doar WCET din punct de vedere al timpilor de execuție. Analiza WCET determină cel mai lung timp posibil pentru execuția unui program. Estimările corecte ale WCET reprezintă un aspect foarte important în cadrul dezvoltării programatoarelor (care programează în timp procesele).

$$C\_{i}^{p}=\sum\_{j=1}^{b}C\_{i}^{p}\left(U\_{i,j}\right)+\sum\_{j=1}^{g}C\_{i}^{p}\left(V\_{i,j}\right)+\sum\_{j=1}^{c}C\_{i,nv j}$$

**Ecuația pentru calculul WCET al unui proces τ­I** [7]

 Ecuația pentru calculul WCET este o funcție care ține cont de cele „b” tampoane de intrare, cele „g” variabile globale de stare și cele „c” secțiuni „nevolatile” ale procesului. Pentru obținerea exactă a valorii WCET, trebuie găsit maximul funcției de mai sus.[7]

 După terminarea analizei temporale trebuie ales un sistem de programare în timp a proceselor. Pentru determinarea ordinii de execuție a proceselor (programarea lor în timp), există mai multe metode, printre care programarea cu priorități fixe și programarea cu priorități dinamice.

 **Programarea cu priorități fixe (Fixed Priority Scheduler)** presupune stabilirea priorității fiecărui proces al sistemului în etapa de proiectare. Acest tip de programare în timp este foarte ușor de implementat într-un sistem de operare în timp real (RTOS), iar testele de fezabilitate sunt ușoare și eficiente. Totuși ca să poată fi folosit acest tip de programare, toate sarcinile care trebuie îndeplinite trebuie să fie considerate periodice.

 În utilizarea acestui tip programare în timp a proceselor, după acordarea diverselor priorități proceselor, trebuie, la orice moment de timp, să fie rulat procesul în stare de așteptare care are prioritatea cea mai mare. [6]

 În continuare vom lua un exemplu cu două procese, T1 și T2.

Procesul T1 are:

* Durata de 6 unități de timp;
* Termenul limită stabilit la fiecare 10 unități de timp;
* Prioritate mai mare.

Procesul T2 are:

* Durata de 9 unități de timp;
* Termenul limită stabilit la fiecare 30 unități de timp;
* Prioritate mai mică.



Cazul 1: procesul T2 este împărțit în 3, deoarece procesul T1 avea prioritate mai mare

Ilustrație din [6]

Observăm în acest prim caz că de fiecare dată când procesul T1 este gata de rulare, procesul T2 este pus în așteptare, iar toate termenele limită sunt îndeplinite.



Cazul 2: procesul T2 este lăsat să ruleze până la sfârșit

Ilustrație din [6]

 În cazul al doilea observăm faptul că procesul T1 începe a doua oară abia dupa terminarea procesului T2. Acest lucru duce la depășirea termenului limită, fapt ce atrage după fie răspunsuri eronate, fie erori de sistem.

 **Programarea în timp cu priorități dinamice** este numită ca fiind de tip „Earliest Deadline First” – cel mai apropiat termen limită mai întâi. Acest tip de programare este considerabil mai eficientă decât cea cu priorități fixe, însă este mult mai greu de implementat într-un sistem de operare în timp real. De asemenea, un alt avantaj este dat de faptul că poate trata și procese neperiodice (determinate de evenimente neperiodice).[6]

 În implementarea acestui tip de programare, procesele care au termenul limită cel mai apropiat sunt considerate ca având prioritate mai mare.

 În continuare, vom aborda cazul acelorași două procese, T1 și T2

Cazul 1: procesul T2 este întrerupt atunci când procesul T1 poate fi reluat.

Ilustrație din [6]

 În acest caz, observăm respectarea regulii de bază a programării cu priorități dinamice: după terminarea procesului T1, procesul T2 începe să fie rulat, dar la momentul 10, când procesul T1 poate fi reluat și are prioritate mai mare (termenul limită nou al lui T1 e mai apropiat), procesul T2 este întrerupt. La momentul 20 poate fi rulat fie procesul T1 fie procesul T2, ele având același termen limită și putând fi încadrate în acel interval de timp.

Cazul 2: procesul T2 nu este întrerupt atunci când procesul T1 poate fi reluat.

Ilustrație din [6]

 În al doilea caz, la momentul de timp 10 procesul T1 poate fi reulat, dar procesul T2 nu este întrerupt, deși T1 are prioritate mai mare, fapt ce atrage depășirea termenului limită a procesului T1.[6]

**5.Bibliografie**

1. http://en.wikibooks.org/wiki/Embedded\_Control\_Systems\_Design/Design\_criteria
2. http://ifs.host.cs.st-andrews.ac.uk/Books/SE7/Presentations/PDF/ch15.pdf
3. http://en.wikipedia.org/wiki/Real-time\_computing
4. http://en.wikipedia.org/wiki/Embedded\_system
5. <http://en.wikipedia.org/wiki/Embedded_software>
6. <http://beru.univ-brest.fr/~singhoff/ENS/USTH/sched.pdf>
7. Approximation Techinques For Timing Analysis Of Complex Real-Time Ebedded Systems ([http://www.diva-portal.org/smash/get/diva2:352398/FULLTEXT02](http://www.diva-portal.org/smash/get/diva2%3A352398/FULLTEXT02) )
8. Patterns for Distributed Embedded Control System Software Architecture (<http://www.cs.tut.fi/~ohar/kirjallisuutta/patterns-vikingplop.pdf>)
9. Software Engineering – Ninth Edition, Ian Sommerville