**Universitatea Politehnica Bucureşti**

**Facultatea Electronică,Telecomunicaţii şi Tehnologia Informaţiei**

Schimbul de mesaje intre firele de executie

**Masterand Caracuda Maria Cecilia**

**Grupa I-IISC**

**2016**

 Cuprins

[Introducere 2](#_Toc460962834)

[1. Schimbul de mesaje la Android OS 2](#_Toc460962835)

[1.2 Tratarea evenimentelor la Android OS 6](#_Toc460962836)

[2. Comunicarea între procese la BlackBerry 10 7](#_Toc460962837)

[2.1 Pasarea sincronă a mesajelor la BB 10 OS 8](#_Toc460962838)

[2.2 Tratarea evenimentelor 9](#_Toc460962839)

[Bibliografie 11](#_Toc460962840)

# Introducere

In aplicatiile multithreaded, sarcinile pot rula în paralel și colabora pentru a produce un rezultat. De aceea, firele trebuie să fie în măsură să comunice pentru a permite o procesare corespunzatoare a datelor si a se obtine o performanta mai ridicata intr-un timp cat mai scurt.

# Schimbul de mesaje la Android OS

Platforma Android definește propriul mecanism de pasare a mesajelor pentru comunicarea între firele de execuție. Firul serverului va putea transfera sarcini lungi prin trimiterea de mesaje de date pentru a fi prelucrate în fire de execuție în fundal. Mecanismul de transmitere a mesajelor este un model de tip consumator-producator care nu presupune blocări, în cazul în care nici firul producător, nici firul consumator nu se vor bloca în timpul mesajului de hand-off.[1]

Mecanismul de manipulare al mesajelor este fundamental în platforma Android și API-ul se află în pachetul android.os, cu un set de clase care implementează funcționalitatea prezentate în figura urmatoare.[1]

Fig 1. Clasele gestioneaza mesajele

Sursa:https://www.safaribooksonline.com/library/view/efficient-android-threading/9781449364120/ch04.html

Mesajele sunt introduse de firul de execuție producător, și sunt procesate de firul consumator, după cum este ilustrat în figura următoare.

Fig 2 Modelul producator-consumator

Sursa:https://www.safaribooksonline.com/library/view/efficient-android-threading/9781449364120/ch04.html

Insert
Firul producător introduce mesaje din coadă folosind handlerul conectat la firul consumator.

Retrive
Looper-ul, se execută în firul consumator și preia mesajele din coada într-o ordine secvențială.

Dispach

Handler-ul este responsabil pentru procesarea mesajelor din firul consumator. Un fir de execuție poate avea mai multe instanțe ale handler-ului pentru procesarea mesajelor; Looper-ul se asigură că mesajele sunt expediate corect de la Handler.

Looper-ul
Clasa android.os.Looper se ocupă de expedierea mesajelor din coadă de la handler-ele asociate. Toate mesajele care au trecut bariera de expediere, sunt eligibile pentru expediere de către Looper. Atâta timp cât coadă are mesaje eligibile pentru expedieri, Looper-ul se va asigura că firul consumator primește mesaje. Când nu există mesaje care au trecut bariera de expediere, firul consumator se va bloca până când un mesaj a trecut de bariera de expediere.[2]

Firul consumator nu interacționează cu coada de mesaje direct pentru a prelua mesajele. În schimb, se adaugă o coadă de mesaje firului de execuție când Looper-ul a fost atașat. Looper-ul gestionează coada de mesaje și facilitează trimiterea de mesaje la firul consumator.

În mod implicit, numai firul serverului are un Looper; firele create în aplicații necesită obținerea unui Looper asociat în mod explicit. Când Looper-ul este creat pentru un fir de execuție, acesta este conectat la o coadă de mesaje. Looper-ul acționează ca intermediar între coada si firele de execuție. [2]

Handler-ul

Până în prezent, accentul a fost pus pe componentele interne de comunicație ale firelor de execuție în Android, dar o aplicație interacționează în principal cu clasa android.os.Handler. Acesta este un API format din două părți, care manipulează atât introducerea mesajelor în coadă cât și prelucrarea mesajului. Acesta este apelat atât la firul producător cât și de firul de consumator, folosit în general pentru:

* Crearea mesajelor
* Inserarea mesajelor în coadă
* Procesarea mesajelor din firul consumator
* Managerierea mesagelor în coadă

Coada de mesaje este reprezentata de clasa android.os.MessageQueue. Aceasta este construita cu mesaje legate, care constituie o listă direcțională. Firul producător insereaza mesaje care ulterior vor fi expediate către consumator. Mesajele sunt sortate în funcție de marcajele de timp. Mesajul în așteptare cu cea mai mică valoare a marcajului de timp este primul în linie pentru a fi expediat către consumator. Cu toate acestea, un mesaj este expediat numai daca valoarea este mai mică decât timestamp-ul ora curentă. În cazul în care nu, expedierea va aștepta până cand ora actuală a depasit valoarea amprentei de timp.[2]

Acest lucru putem observa din figura urmatoare.



Fig 3. Prioritizarea mesajelor din coada

Sursa:https://www.safaribooksonline.com/library/view/efficient-android-threading/9781449364120/ch04.html

În cazul în care niciun mesaj nu a trecut bariera de expediere atunci când Looper-ul este gata pentru a prelua mesajul următor, consumatorul se blocheaza. Execuția este reluată imediat ce un mesaj trece de bariera de expediere(bariera de timp).[2]

Producătorii pot insera mesaje noi în coada de așteptare, în orice moment și în orice poziție din coadă. Poziția de inserție în coada de așteptare se bazează pe valoarea amprentei de timp. În cazul în care un nou mesaj are cea mai mică valoare a amprentei de timp, comparativ cu mesajele în așteptare din coadă, acesta va ocupa prima poziție în coada de așteptare, care este urmatorul ce va fi expediat. Inserțiile respectă întotdeauna ordinea de sortare a amprentei de timp. Mesajul de inserare este discutat în continuare în Handler.

Ciclul de viață al unui mesaj este urmatorul: producătorul creează mesajul, iar în cele din urmă este procesat de către consumator. Această descriere este suficientă pentru cele mai multe cazuri de utilizare, dar atunci când apare o problemă, o înțelegere mai profundă a manipulării mesajului este de neprețuit. Să ne aruncăm o privire în ceea ce se întâmplă de fapt cu mesajul în timpul ciclului său de viață, care poate fi împărțită în patru stări principale prezentate în figura. [2]



Fig 4. Starile unui mesaj

Sursa:https://www.safaribooksonline.com/library/view/efficient-android-threading/9781449364120/ch04.html

Inițializarea

In starea de inițializare, un obiect mesaj cu starea transformabila a fost creat și, în cazul în care acesta este un mesaj de date, populat cu date. Aplicatia este responsabila pentru crearea obiectului mesaj.

In asteptare

Mesajul a fost inserat în coada de firul producător, și așteaptă să fie expediate la firul consumator.

Expediat

În această stare, Looper-ul a preluat și a îndepărtat mesajul din coada. Mesajul a fost trimis la firul consumator și este în prezent în curs de procesare.

Reciclat

La acest punct din ciclul de viata, starea mesajului este ștearsă, iar instanța este returnata pool-ului de mesaje. Looper-ul se ocupă de reciclarea mesajului atunci când acesta a terminat executarea pe firul consumator. [2]



Fig 5. Preluarea mesajelor din coada

Sursa:https://www.safaribooksonline.com/library/view/efficient-android-threading/9781449364120/ch04.html

## 1.2 Tratarea evenimentelor la Android OS

Există, trei concepte legate de managementul evenimentelor la Android OS:

Events Listeners – reprezintă o interfață în clasa View, care conține o singură metodă de apel invers. Aceste metode vor fi apelate de framework-ul Android, atunci când View, la care a fost înregistrat Listener-ul este declanșat de interacțiunea utilizatorului cu interfața de utilizare.

Event Listener Registration – reprezintă procesul prin care o Handler-ul evenimentului devine înregistrat într-un procesare a evenimentelor, astfel încât Handler-ul este apelat atunci când Event Listener-ul declanșează evenimentul.

Event Handlers – Atunci când are loc un eveniment și a fost înregistrat un Event Listener pentru acesta, Tvent Listener-ul apelează Event Handler-ul, care este metoda care se ocupă de fapt de eveniment.[2]

# Comunicarea între procese la BlackBerry 10

Comunicarea între procese joacă un rol fundamental în procesul de transformare a microkernel-ului QNX Neutrino de la un nucleu în timp real încorporat în sistemul de operare POSIX pe scară largă. Pe măsură ce diverse procese pentru diferite servicii se adaugă la microkernel, IPC este liantul care leagă aceste componente într-o unitate coerentă. Deși schimbul de mesaje are de forma primitivă a IPC-ului în BlackBerry 10 OS, multe alte forme sunt de asemenea disponibile. Cu excepția cazului în care se menționează altfel, aceste alte forme de IPC sunt construite în mesajul nostru nativ de pasare a mesajelor. Strategia este de a crea un serviciu IPC simplu, robust a cărui performanță poate fi controlată printr-o cale de cod simplificată în microkernel; multe alte servicii IPC pot fi apoi implementate plecând de la acestea.[3]

BlackBerry 10 OS oferă cel puțin următoarele forme de IPC:

|  |  |
| --- | --- |
| Serviciu: | Implementat în: |
| Pasarea mesajelor | Kernel |
| semnalele | Kernel |
| Memorie partajată | Managerul de procese |
| Pipe-uri | Proces extern |
| FIFO | Proces extern |

Proiectantul poate selecta aceste servicii pe baza cerințelor de lățime de bandă, nevoia de așteptare a proceselor în coadă, transparența rețelei, și așa mai departe. Compromisul pot fi greu de ales, dar flexibilitatea este foarte utilă.

Ca parte a efortului de inginerie care a intrat în definirea microkernel-ului QNX Neutrino, accentul pe transmiterea de mesaje ca IPC fundamentale primite a fost pus în mod deliberat. Ca o formă de IPC, transmiterea de mesaje (așa cum este implementat în MsgSend (), MsgReceive (),și MsgReply ()), este sincronă și se face prin copierea datelor.[3]

## 2.1 Pasarea sincronă a mesajelor la BB 10 OS

Sincronizarea mesajelor este principala formă de IPC în BlackBerry 10 OS. Un fir care face o intrsuctiune MsgSend () la un alt fir de execuție (care ar putea fi într-un alt proces) este blocat până când firul țintă face un MsgReceive (), procesează mesajul și execută un MsgReply (). În cazul în care un fir execută un MsgReceive (), fără un mesaj trimis anterior în așteptare, acesta se blochează până când un alt fir execută MsgSend ().[3]

În BlackBerry 10 OS, un fir de execuție de tip server intră de obicei într-o buclă, așteaptând să primească un mesaj de la un fir de execuție client. Așa cum este descris mai devreme, un fir - indiferent dacă este client sau server - se află în starea READY dacă poate folosi CPU. S-ar putea să nu primescă timp CPU, din cauza politicii, prioritații sau planificării sale sau a altor fire, dar acesta nu este blocat.[3]



Fig 6. Firul de execuție al unui proces client

Sursa [*https://developer.blackberry.com/native/documentation/*](https://developer.blackberry.com/native/documentation/)

* În cazul în care firul clientului solicită MsgSend (), iar firul serverului nu a apelat încă MsgReceive (), atunci firul TRIMITE al clientului devine blocat. În cazul în care firul serverului aplelează MsgReceive (), kernel-ul schimbă starea firul REPLY al clientului să fie blocat, ceea ce înseamnă că firul serverului a primit mesajul, iar acum trebuie să răspundă. În cazul în care firul serverului aplelează MsgReply (), firul clientului devine READY.
* În cazul în care firul clientului solicită MsgSend (), iar firul serverului este deja blocat pe MsgReceive (), firul REPLY al clientului devine imediat blocat, nemaitrecând prin stadiul în care SEND este blocat.
* În cazul în care firul serverului nu reușește, iese, sau dispare, firul clientului devine READY, cu MsgSend (), indicând o eroare.

 
 În continuare, să considerăm firul serverului:

* În cazul în care firul serverului apelează MsgReceive (), și nici un alt fir a trimis nimic la el, acesta devine RECEIVE blocat. Atunci când un alt fir trimite la el, firul serverului devine READY.
* În cazul în care firul de server apelează MsgReceive (), și un alt fir a trimis deja la el, atunci MsgReceive () devine instantaneu READY. În acest caz, firul de server devine blocat.
* În cazul în care firul server apelează MsgReply (), aceasta nu devine blocat.[3]

Acest blocaj inerent sincronizează executarea firului care trimite mesajul, întrucât actul de a solicita ca datele să fie transmise cauzează de asemenea, ca firul care face trimiterea să fie blocat, iar firul de recepție să fie programat pentru execuție. Acest lucru se întâmplă fără a necesita munca explicită de către nucleu pentru a determina care thread urmează sa ruleze următorul (așa cum ar fi cazul cu cele mai multe alte forme de IPC). Execuția și pasarea datelor balează usor de la un context la altul.[3]

MsgReply () vs MsgError ()

Funcția MsgReply () este folosită pentru a returna o stare și zero sau mai multi baiți pentru client. MsgError (), pe de altă parte, este folosit pentru a returna numai o stare de client. Ambele funcții scot clientul din starea MsgSend ().

## 2.2 Tratarea evenimentelor

Un progres semnificativ în proiectarea nucleu pentru BlackBerry 10 OS este subsistemul de manipulare evenimentelor. POSIX și extensiile sale în timp real definesc o serie de metode de notificare asincrone (de exemplu, semnalele UNIX care nu se formează în coadă sau care trec de date, semnale în timp real POSIX, care se pot forma în coadă și să treacă de date, și așa mai departe). Kernel-ul definește, de asemenea, la BlackBerry 10 tehnici suplimentare, specifice de notificare, cum ar fi de impulsuri. Punerea în aplicare a tuturor acestor mecanisme de evenimente ar fi putut consuma cod si spațiu semnificative, astfel încât strategia de implementare a fost de a construi toate aceste metode de notificare într-o singur subsistem bogat de manipulare a evenimentelor.[3]

Un beneficiu al acestei abordări este că capacitățile exclusive pentru o tehnică de notificare pot deveni disponibile pentru alte persoane. De exemplu, o aplicație poate aplica aceleași servicii de ordonat în coadă a semnalelor în timp real POSIX la semnalele UNIX. Acest lucru poate simplifica implementarea robustă de manipulare de semnale în cadrul aplicațiilor.

Evenimentele întâmpinate de un fir de execuție pot veni de la oricare dintre cele trei surse de:

* Un eveniment MsgDeliverEvent()apelat de un fir de execuție
* De un handler de întreruperi
* Expirarea unui temporizator

Evenimentul in sine poate fi de diferite tipuri: impulsuri ale sisteului de operare BlackBerry 10, întrerupe, diferite forme de semnale și evenimente de deblocare forțate. Deblocarea este un mijloc prin care un fir poate fi eliberat dintr-o stare de blocare în mod deliberat, fără ca vreun eveniment sa fie de fapt livrat.

Având în vedere această multitudine de tipuri de evenimente, și aplicații care necesită abilitatea de a solicita oricare tehnică de notificare asincronă se potrivește cel mai bine nevoilor lor, ar fi ciudat să se solicite ca procesele de server sa transporte cod pentru a sprijini toate aceste opțiuni.

În schimb, firul clientului poate da o structură de date, sau cookie, serverului ca acesta să stea în hold până mai târziu. În cazul în care serverul trebuie să notifice firul clientului, acesta apelează MsgDeliverEvent () și microkernel-ul stabilește tipul de eveniment codificat în interiorul cookie din firul clientului.[3]

Fig 7. Comunicatie client server

Sursa [*https://developer.blackberry.com/native/documentation/*](https://developer.blackberry.com/native/documentation/)

Funcția ionotify () este un mijloc prin care un fir al clientului poate solicita livrarea asincronă de evenimente. Multe dintre serviciile asincrone POSIX (de exemplu, partea de client a funcției select ()) sunt construite peste ionotify (). Atunci când se efectuează I / O pe un descriptor de fișier (fd), firul poate alege să aștepte pentru un eveniment de I / O pentru a se finaliza (pentru cazul write()), sau pentru ca datele să soseasca (pentru cazul read()). Mai degrabă, decât ca blocul firelor de execuție în procesul managemrntului de resurse care servește la funcțiile de citire și de scriere, ionotify () poate permite firului clientului să posteze un eveniment în managerul de resurse cum că firul de client ar dori să primească o notificare atunci când de produce un eveniment I/O. Această așteptare permite firului să continue execuția și să anumite evenimente.[3]

Funcția select () este implementată folosind o notificare I/O și permite unui fir să se blocheze și să aștepte o anumită înșiruire de evenimente I/O din mai mulți descriptori de fișiere în timp ce continuă să răspundă la alte forme de IPC.

# Bibliografie

1. Anders Goransson, “*Efficient Android Threading*”, June 2014
2. <https://zonait.tv/sisteme-de-operare-alternative-la-android-si-ios/>, accesat la data de 11.1.2016
3. [*https://developer.blackberry.com/native/documentation/*](https://developer.blackberry.com/native/documentation/)*,* accesat la data de 11.11.2015