

## 4.4. Kernel-uri de executie: micro-, exo-, monolitic

În cadrul sistemelor de calcul se pot desfășura procesele care utilizează memoria sistemului de calcul care vor cere aceasta memorie de la sistemul de operare – acesta însă nu le alocă adrese în spațiul fizic, real, ci va face legătura între paginile de memorie alocate și paginile într-un spațiu virtual.

Cel mai des, sistemul de operare va lăsa în spațiul virtual zone fără echivalent în spațiul real - când procesorul întâlnește astfel de adrese el va genera o întrerupere software, semnalizându-i sistemului de operare faptul că memoria invalidă a fost accesată. Acest procedeu este probabil cel mai util pentru depanarea programelor și urmărirea eventualelor greșeli de acces la rularea normală a procesului.

Deoarece regulile sistemului de calcul nu sunt neapărat identice pentru toate sistemele în parte, sistemul de operare ascunde complexitatea hardware-ului în spatele unor interfețe standardizate. Tradițional, interfața de alocare de memorie este cât se poate de simplistă:

*malloc(size): memory allocation* - alocă o zonă de memorie de *size* octeți

*free(address):* eliberează zona de memorie de la adresa *address*

În spatele acestei interfețe simpliste se află algoritmi complecși de alocare a paginilor de memorie, de mapare a paginilor de memorie din spațiul real în spațiul virtual. De aceea fiecare alocare de memorie realizată de către un program utilizator va schimba contextul din spațiul protejat, virtual al procesului în spațiul real, al kernelului.

Acest detaliu este foarte important pentru că această schimbare de context este extrem de costisitoare în termeni de performanță a sistemului de operare. Cu alte cuvinte, alocarea de memorie în cazul unui proces utilizator va urma pașii de mai jos:

1. apel *malloc* din biblioteca standard
2. biblioteca standard apelează funcția sistem de alocare de memorie
3. se trece în modul *kernel*
4. se identifică paginile necesare pentru alocarea de memorie
5. se creează corespondențele între pagini și zona de memorie virtuală
6. se reîntoarce în modul *user*
7. se returnează adresa virtuală către procesul apelant

Din punct de vedere al performanței, trecerea între modul *kernel* și modul *user* este o problemă nu numai pentru că trecerea între cele două nivele de permisiuni este costisitoare ca timp, dar pentru marea majoritate a sistemelor de operare, toate celelalte fire de execuție sunt suspendate. Mai mult, înainte sau după trecerea în modul *kernel* este foarte posibil ca execuția firului să fie preempțată și firul să reprimească controlul mult mai târziu. Pentru sistemele moderne, cu capacitate de calcul foarte mare, această problemă pare insignifiantă.

### 4.2.1 Restricții de memorie în mediul EMBEDDED

Nu același lucru se poate spune despre spațiul EMBEDDED. Sistemele EMBEDDED sunt sisteme de calcul minimale, cu resursele calculate pentru optimizarea costului de producție, nu a performanțelor. Cel mai des, sistemul e foarte limitat ca putere de procesare și putere de



**Structuri hardware si algoritmi specifici microsistemelor EMBEDDED**

memorare, orientându-se pe minimalizarea resurselor folosite pentru a duce la bun sfârșit scopul pentru care a fost creat.

În variantele de bază, sistemele EMBEDDED nu au parte de un sistem de operare. Problema

pentru astfel de sisteme este costul de programare a dispozitivului - este mult mai scump să programezi funcționalități pe care în mod normal le întâlnești într-un sistem destinat acestui gen de operare.

În variantele moderne, un sistem de operare standardizat e folosit pentru a face managementul dispozitivului ce se dorește programat. Un astfel de sistem este Linux - care oferă posibilitatea de a selecta componentele care se doresc incluse în interiorul unei imagini la construcția sistemului, sau WindowsCE care oferă posibilitatea alegerii nu numai a unor componente de nivel jos

(*kernel*) dar chiar și componente de nivel înalt, precum *browser* sau *player* multimedia. O altă variantă este ca dispozitivul să ofere o mașină virtuală standardizată, precum Java sau NET (mai rar).

De ce se alege un sistem standardizat? Folosirea unui astfel de sistem prezintă dezavantaje,

precum o amprentă de memorie (mult) mai mare și un *overhead* pentru toate acțiunile tipice (precum alocarea de memorie). Dar avantajele sunt mult mai mari; motivul principal pentru care am prefera să folosim un sistem standardizat este abilitatea de a testa software-ul înainte de a avea hardware-ul și faptul că folosim componente standardizate care ar trebui implementate de către programatori în cazul sistemului creat specific pentru un anumit dispozitiv. De aceea, în mediul EMBEDDED vom simți mult mai puternic scăderile de performanță provocate de trecerea între contextul *user* și contextul *kernel*. Dacă aplicația va face multe alocări/dealocări de memorie, această operație va deveni ea însăși și un element de luat în seamă în evaluarea performanțelor și a timpilor de execuție.

O altă problemă mai gravă este fragmentarea memoriei alocate. Maparea paginilor de memorie salvează parțial mediul EMBEDDED de această problemă. Uneori însă, este nevoie de alocarea unei zone de memorie contigue - de exemplu alocarea unei zone de memorie care să țină imagini pregătite pentru operații accelerate în hardware. În acest caz, datorită unui plasament greșit din partea sistemului de operare, este posibil ca o astfel de zonă de memorie să nu fie disponibilă decât prin mutarea unor pagini din memorie, iar acest lucru este puțin probabil în cazul sistem care nu deține suport pentru memorie externă (hard-disk pentru *swapping*). În plus, urmărirea memoriei pe un dispozitiv minimal ar putea fi foarte dificilă doar bazându-ne pe suportul oferit de sistemul de operare - de aceea am putea să folosim o bibliotecă de alocare chiar și numai pentru a urmări mult mai eficient tipurile de obiecte alocate și de a le controla dacă memoria folosită de obiecte nu este uitată ne alocată.

Așadar calitățile pe care le dorim de la un alocator de memorie în mediul EMBEDDED sunt:

1. fragmentare minimă a memoriei;
2. cât mai puține *switch*-uri de context;
3. cât mai puține *lock*-uri pe obiecte de sincronizare (ideal ar fi ca memory-manager-ul să nu fie afectat de lucrul cu mai multe *thread*-uri);
4. să ofere posibilitatea investigării obiectelor alocate și a posibilelor *memory leak*-uri.



UNIUNEA EUROPEANĂ

MINISTERUL MUNCII, FAMILIEI ȘI  
PROTECȚIEI SOCIALE  
AMFOSDRUFONDUL SOCIAL EUROPEAN  
POSDRU  
2007-2013INSTRUMENTE STRUCTURALE  
2007-2013

**Bibliografie:**

1. Istvan Sztojanov, Sever Pașca, Elisabeta Buzoianu, Aplicații hardware și software cu microcontrolerul PIC12F675, Editura Cavallioti, ISBN 978-973-7622-54-9, Bucuresti 2008
2. Istvan Sztojanov, Alexandru Vasile, Elisabeta Buzoianu, Sever Pașca, *Programarea microcontrolerelor din familia Intel, Aplicații practice hardware cu 80C552*, Editura Man-Dely, ISBN 973-85681-5-3, București 2004.
3. <http://vega.unitbv.ro/~romanca/EmbSys/>
4. <http://facultate.regielive.ro/cursuri/electronica/>
5. [www.microcip.com](http://www.microcip.com)
6. Andrei Drumea, Teza de doctorat, UPB 2009
7. Dorin Lazăr, Optimizarea alocării de memorie în C/C++ pentru spațiul EMBEDDED  
Universitatea Transilvania, Brașov. 2008



UNIUNEA EUROPEANĂ



MINISTERUL MUNCII, FAMILIEI ȘI  
PROTECȚIEI SOCIALE  
AMFOSDRU



FONDUL SOCIAL EUROPEAN  
POSDRU  
2007-2013



INSTRUMENTE STRUCTURALE  
2007-2013