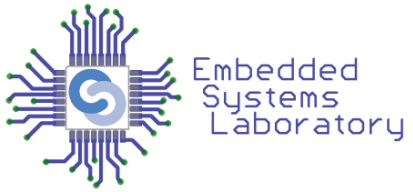


Sisteme Încorporate

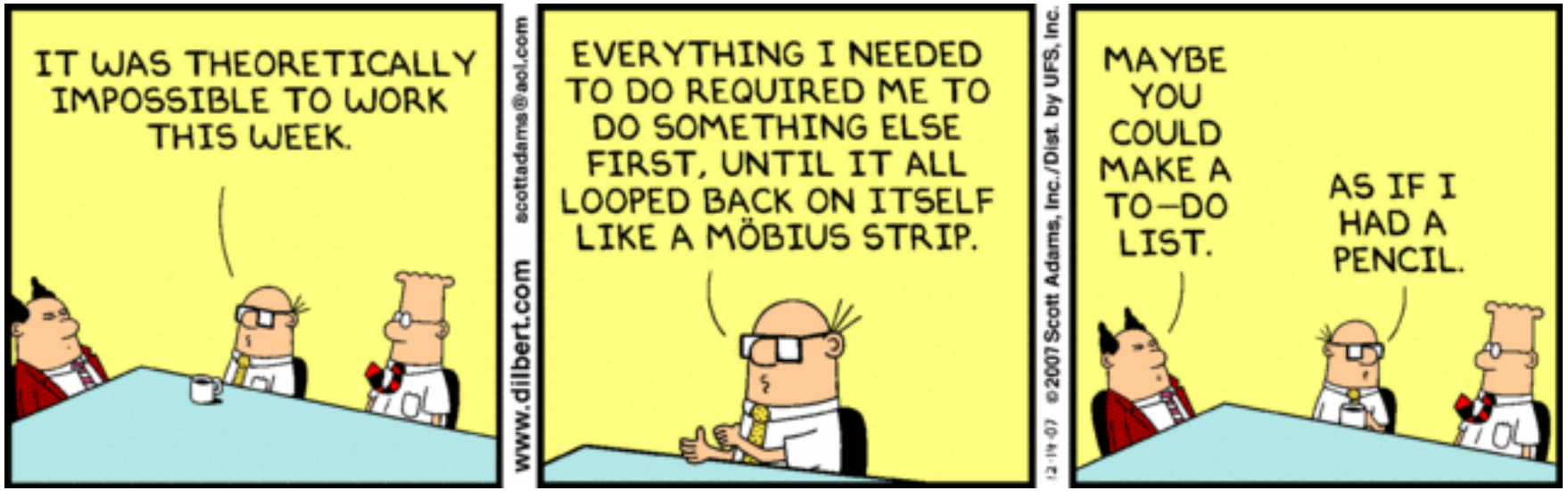
Cursul 9

Software de timp real

Sisteme de operare de timp real

Planificare

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București



http://dilbert.com/strips/comic/2007-12-14/

Definiție: Sisteme de timp real

- Un sistem embedded care monitorizează, răspunde la stimuli sau controlează mediul extern în timp real (sisteme reactive)
- Exemple:
  - Vehicule (automobil, avion, …)
  - Controlul traficului(autostradă, aerian, căi ferate, …)
  - Controlul proceselor (uzină electrică, chimică, …)
  - Sisteme medicale (terapia prin iradiere, …)
  - Telefonie, radio, comunicații prin satelit
  - Jocuri de calculator

Caracteristici

- Constrângeri de timp/termen limită
  - Corectitudine temporală și funcțională
- Hard deadline
  - Trebuie să respecte termenul întotdeauna
  - Controller pentru traficul aerian
- Soft deadline
  - Trebuie să respecte termenul frecvent
  - Decoder MPEG
- Concurență (procese multiple)
  - Face față la semnale multiple de intrare și ieșire
- Fiabilitate
  - Cât de des se defectează sistemul
- Toleranța la defecte
  - Recunoașterea și tratarea erorilor și defectelor
- Sisteme Critice
  - Cost mare al unui defect
  - Sistem hard real time ⇒ sistem critic

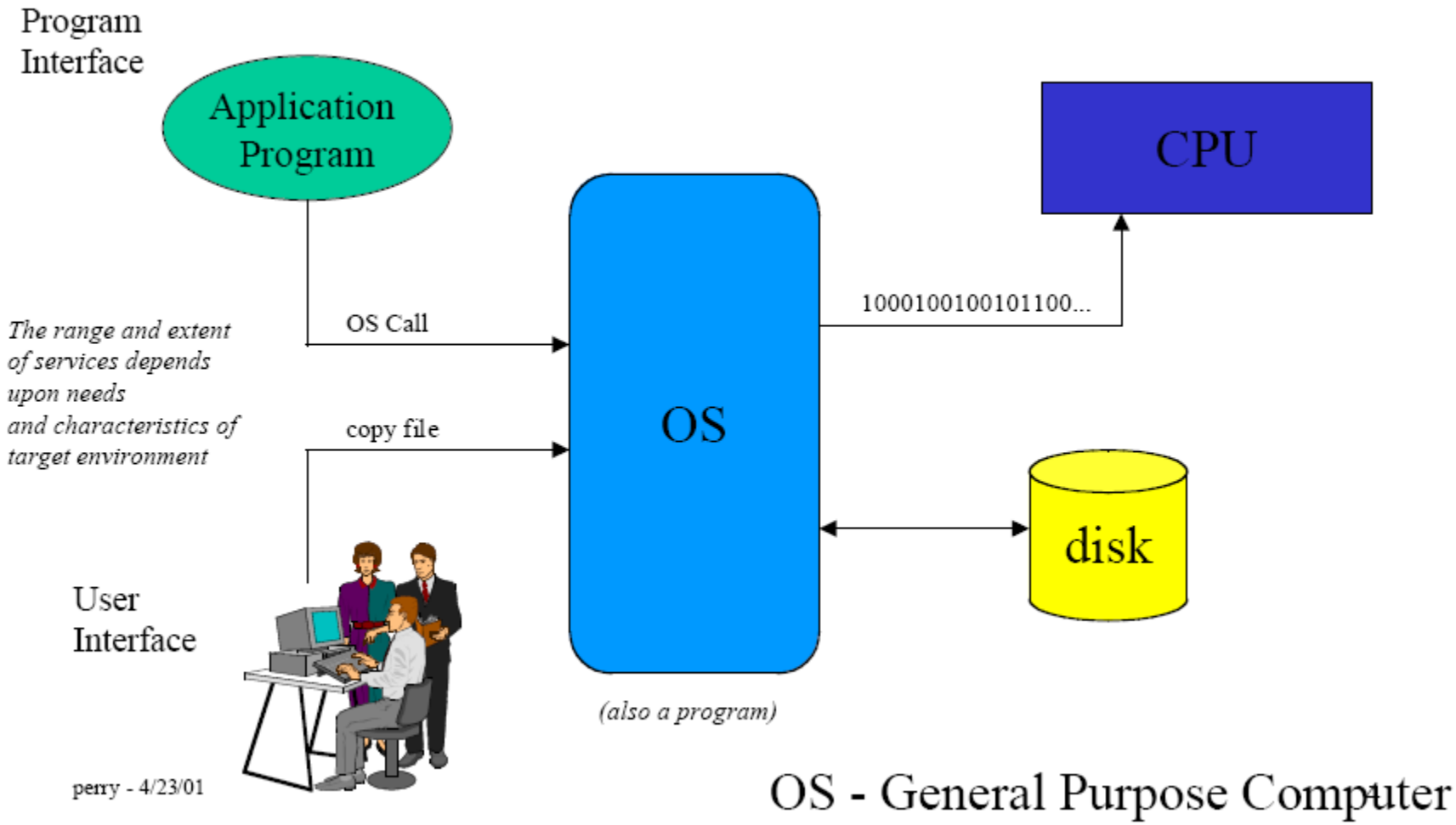
- Abordări tipice:
  - Sincron
    - O singură buclă de program
  - Asincron
    - Sistem foreground/background
    - Multitasking

Sisteme de operare

- Ce este un sistem de operare?

- O colecție organizată de extensii software a hardware-ului care îndeplinesc următoarele funcții:
  - Rutine de control pentru operarea sistemului (permit accesul la resursele calculatorului: file-system, I/O, memorie etc)
  - Un mediu pentru execuția de programe

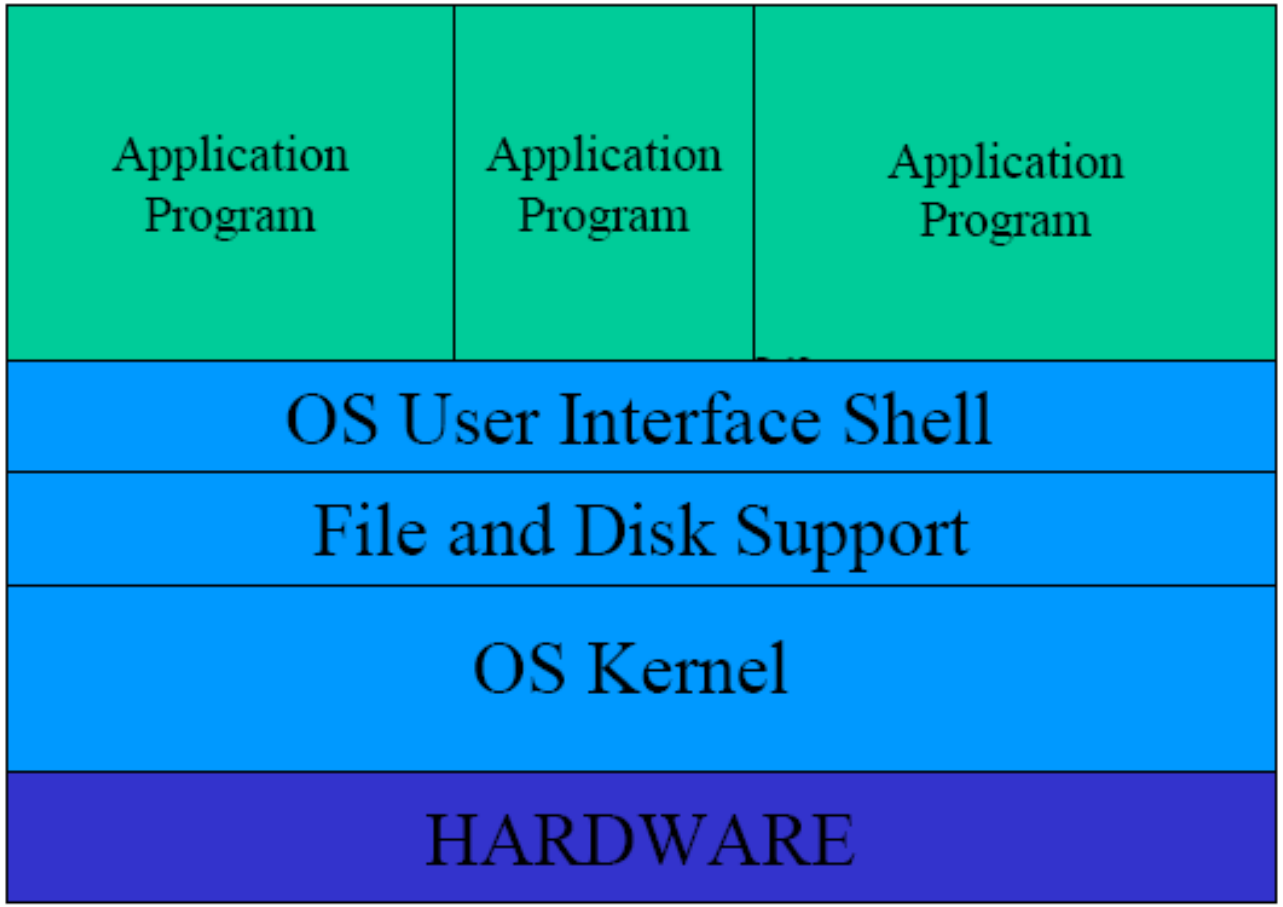
Serviciile unui SO



Sisteme de operare

- Ce face de fapt un sistem de operare?
- Administrează resursele de sistem (procesor, memorie, I/O, etc.)
  - Ține evidența asupra statusului și “proprietarului” fiecarei resurse
  - Decide cine primește resursa
  - Decide cât de mult timp resursa poate fi alocată
- În sisteme cu execuție concurentă
  - Arbitrează și rezolvă conflictele de resurse
  - Optimizează performanțele în contextul utilizatorilor multipli
- Gandiți-vă la un SO ca la proprietarul unei cărți pe care toți studenții de la acest curs trebuie s-o citească
  - Care sunt problemele care apar?

SO – Structura ierarhică

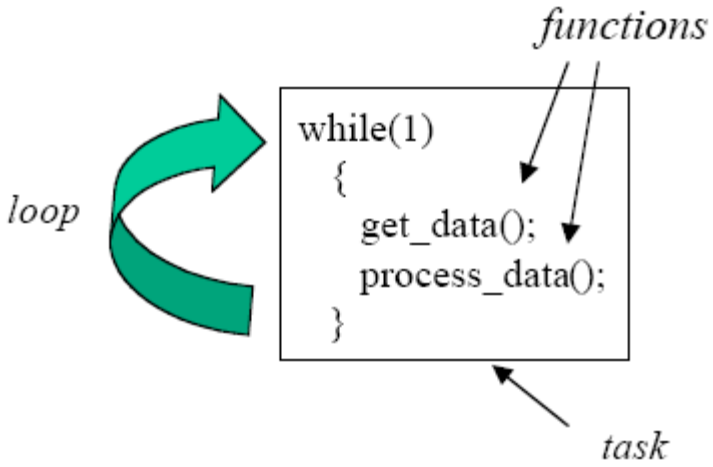


Sisteme de operare

- Tipuri de sisteme de operare
  - Cele mai simple = kernel de dimensiuni mici pe un procesor embedded
  - Complexe = SO comercial Full-Featured
    - Securitate
    - Utilizatori multipli
    - Suport grafic
    - Suport pentru rețea
    - Drivere comunicație cu o gamă largă de periferice
    - Programe cu execuție concurentă

Taskuri și Funcții

- Un task este un proces repetitiv
  - Buclă infinită
  - Conceptul de baza în sistemele de operare de timp real (RTOS).



- O funcție este o procedură care poate fi apelată. Aceasta ruleaza apoi întoarce o valoare la terminarea execuției.
  - process_data();
  - int add_two_numbers(int x, int y);

RTOS

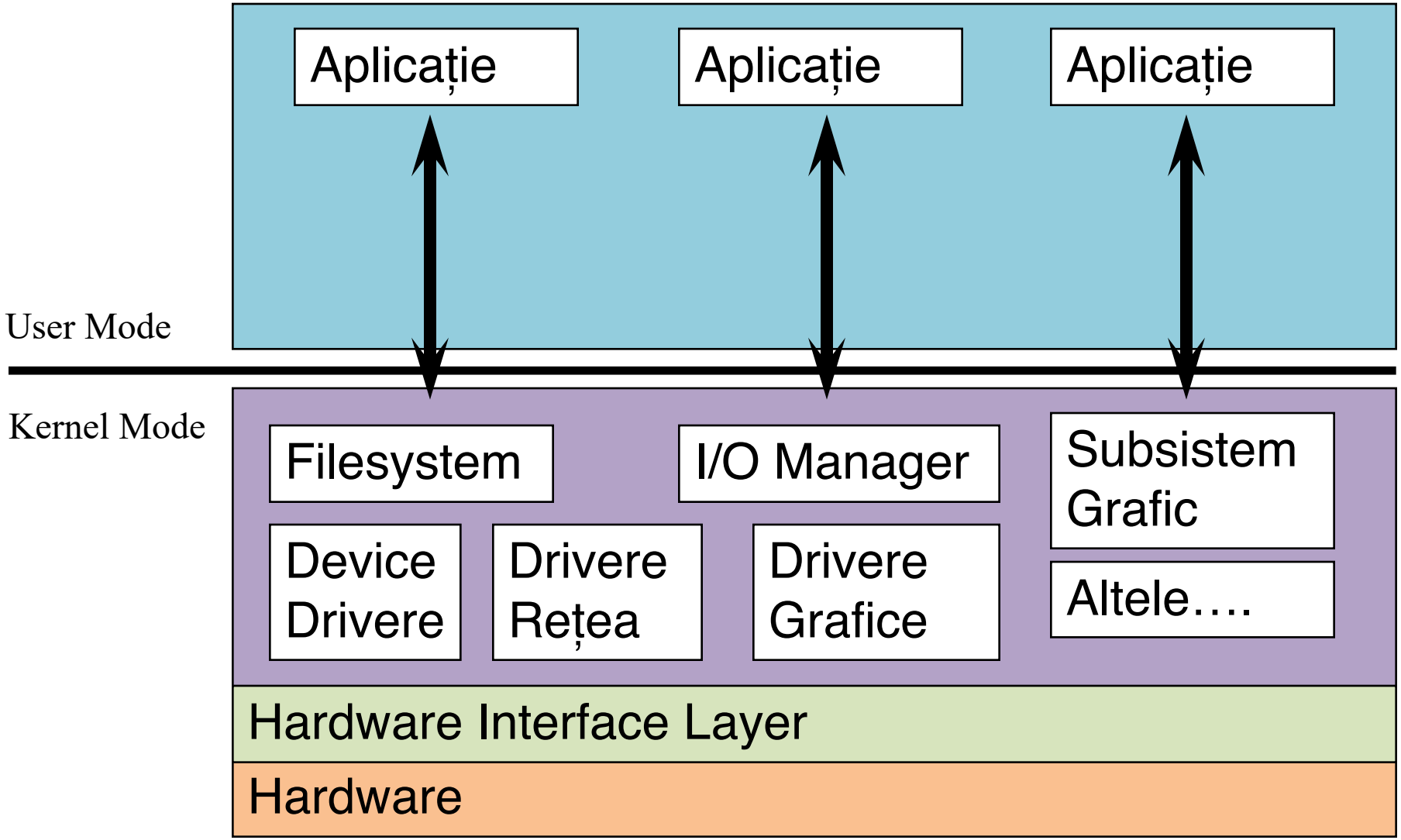
- În majoritatea cazurilor, RTOS = OS Kernel
  - Un sistem embedded este proiectat pentru un singur scop așa că majoritatea funcționalităților unui SO comercial sunt redundante (consolă, interfața grafică, suport tastatură, mouse etc.).
  - RTOS permite controlul ferm asupra resurselor sistemului
    - Nu există procese de background inutile
    - Număr maxim de task-uri care pot rula pe sistem
  - RTOS permite controlul temporizării proceselor
    - Manipularea priorității task-urilor
    - Opțiuni de setare a mecanismului de planificare

Implementarea în Real-Time OS

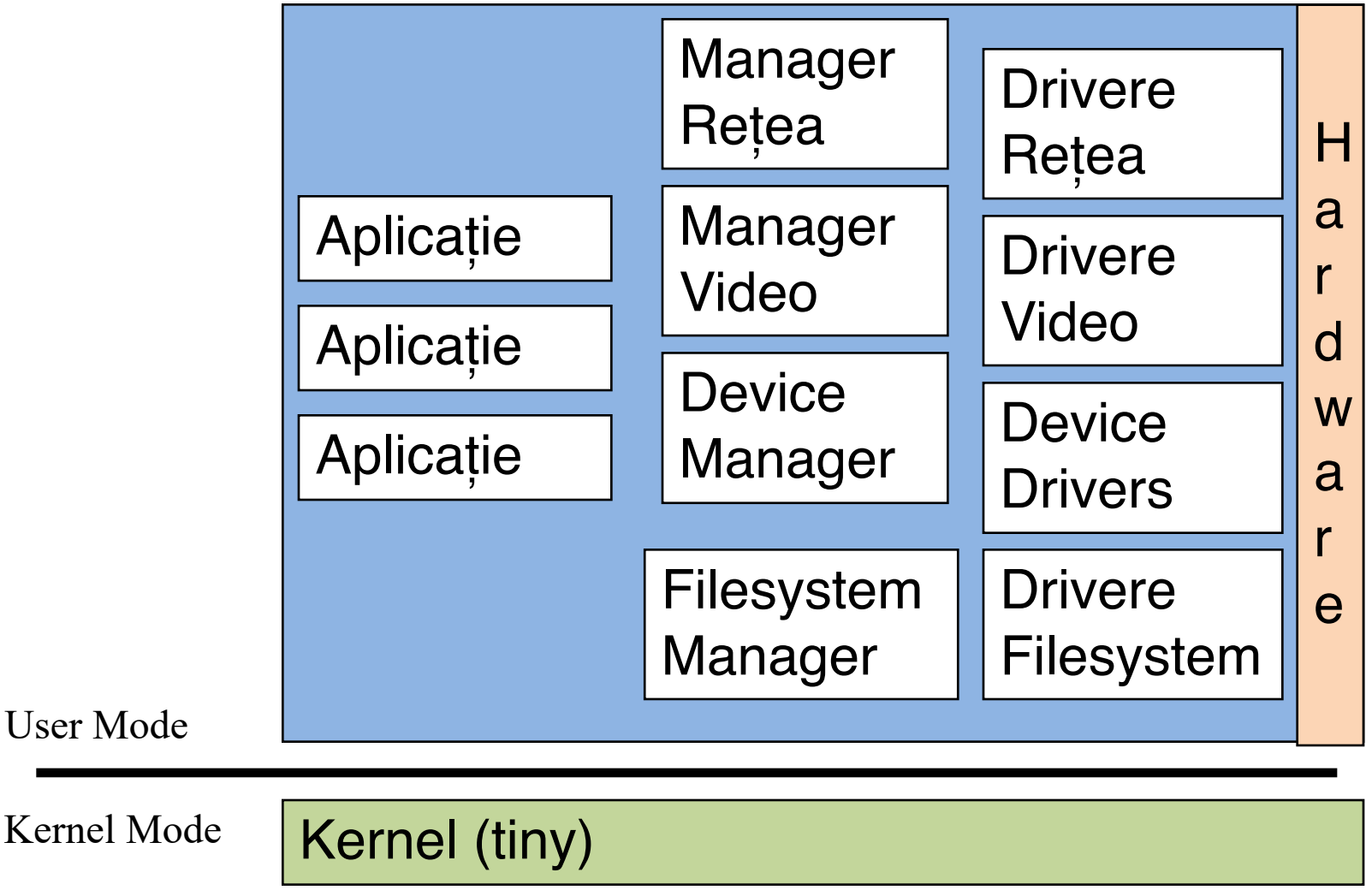
- Nuclee mici și rapide
- Extensii de timp real la sisteme de operare comerciale
- Sisteme de operare experimentale
- Parte din run-time-ul unor limbaje de programare
    - Java (embedded real-time Java)

- Kernel monolitic vs. Microkernel

Organizarea unui kernel monolitic



Organizarea unui microkernel



RTOS - Kernel

- Kernel-ul OS îndeplinește 3 funcții:
  - **Task Scheduler**: Determină care task va rula în cuanta de timp următoare pentru un sistem multitasking.
  - **Task Dispatcher**: Produce informația necesară în contextul pornirii unui task
  - **Intertask Communication**: Implementează un mecanism de comunicație între două procese

RTOS - Kernel

- Dacă ne întoarcem la analogia cu cartea
  - **Task Scheduler**: Cine primește cartea și când?
  - **Task Dispatcher**: Managementul transportului cărții de la o persoană la alta
  - **Intertask Communication**: Ce se întâmplă dacă un student vrea să vorbească cu altul? Doar un singur student poate avea cartea la un moment dat.

RTOS Kernel

- Strategii de design pentru nucleul RTOS
  - Polled Loop Systems
  - Sisteme Interrupt Driven
  - Sisteme Foreground / Background
  - Multi-tasking
  - Full Featured RTOS

Polled Loop System

- Cel mai simplu nucleu real-time
- O singură instrucțiune repetitivă testează un flag care indică dacă un eveniment s-a produs sau nu.
- Nu este nevoie de comunicație intre task-uri sau mecanisme de planificare – avem un singur task.
- Se comportă excelent în cazul canalelor de viteză mare de transmisie a datelor.
  - Evenimentele se petrec la intervale de timp uniforme (și relativ mari)
  - Procesorul se ocupă numai de canalul de date

Exemplu

- Un automat programabil care trebuie să îndeplinească următoarele funcții
  - La fiecare 20ms trebuie să actualizeze ceasul sistemului
  - La fiecare 40ms rulează un modul de control
  - Încă trei module fără constrângeri puternice de timp
    - » Actualizarea ecranului operatorului
    - » Primirea de comenzi de la operator
    - » Înregistrarea unui istoric de evenimente și comenzi

Program cu o singură buclă de control

```
while (1) {
    wait_clock_tick();
    if(time_for_clock) update_clock();
    if (time_for_control) do_control();
    else if (time_for display update) refresh_display();
    else if (time_for_input) get_input();
    else if (time_for_log) save_log();
}
```

- Trebuie ca: $t1 + max(t2, t3, t4, t5) \leq 20\ ms$
  - Se pretează la programe simple, cu numar limitat de funcții și constrângeri

Alt Exemplu

```
int main(void) {
    Init_All();
    for (;;) {
            IO_Scan();
            IO_ProcessOutputs();
            KBD_Scan();
            PRN_Print();
            LCD_Update();
            RS232_Receive();
            RS232_Send();
            TMR_Process();
    }
    // n-ar trebui sa ajunga aici
    printf("Eroare…");
return (0);
}
```

Observații

- Fiecare funcție apelată în bucla infinită se execută independent
- Fiecare funcție trebuie să-și înceteze execuția după un timp rezonabil, indiferent de codul executat.
- Nu se știe frecvența la care se execută bucla principală
  - Frecvența poate varia în funcție de evoluția sistemului și de starea curentă
- Bucla conține și funcții periodice și funcții executabile la anumite evenimente
  - Majoritatea task-urilor sunt event-driven
    - ex. IO_ProcessOutputs este event-driven
    - De obicei au asociate la intrare cozi de mesaje
      - Ex. IO_ProcessOutputs primște evenimente de la IO_Scan, RS232_Receive si KBD_Scan când o ieșire trebuie activată
  - Celelalte sunt periodice
    - Nu răspund la evenimente dar pot avea perioade diferite și își pot schimba în timp perioada de execuție

Observatii (cont.)

- Trebuie implementate niște metode simple de comunicație inter-task
  - Ex. Vrem să nu mai citim intrările după ce s-a apăsat o anumită tastă sau să repornim citirea la altă apăsare
    - Cerere de la KBD_scan() la IO_scan()
  - Ex. Vrem sa restricționam execuția anumitor rutine în funcție de circumstanțe
    - Apare o avalanșă de schimbari de stare la intrările sistemului și legătura RS232 nu poate să le trimită pe toate
    - Reducem perioada IO_scan() a.î. transmisia să fie completă
- Uneori este nevoie să se execute câteva operații simple dar prioritare
  - Ex. Micșorează luminozitatea LCD-ului după un timp de la ultima apăsare, clipește cursorul la poziția curentă pe ecran cu o anumită frecvență fixă.
  - De cele mai multe ori aceste procese nu au funcții dedicate (sunt prea simple) ci sunt executate sincron de o rutină de timer

Polled Loops

- Pro:
  - Foarte simplu de implementat în cod și de depanat
  - Timpul de răspuns este foarte ușor de determinat
- Contra:
  - Poate să cedeze la evenimente în rafală
  - În general, nu are suficiente funcționalități pentru controlul sistemelor complexe
  - Cicli de ceas pierduți, mai ales dacă evenimentul din buclă se petrece foarte rar

Sisteme bazate pe întreruperi

- Ce este o întrerupere?
- Un semnal hardware care inițiază un eveniment
- La recepția unei întreruperi, procesorul:
  - Finalizează instrucțiunea curentă
  - Salvează Program Conter (ca să se întoarcă de unde a pornit)
  - Încarcă în PC adresa de început a rutinei de tratare a întreuperii
  - Execută RTI
- De obicei, sistemele de timp real pot să trateze mai multe întreruperi simultan prin implementarea unui mecanism de priorități
  - Întreruperile pot fi pornite/oprite
  - Întreruperile cu cea mai mare prioritate sunt tratate primele

Implementarea unei rutine event-driven

Fiecare rutină are atașată o coadă de evenimente

Ex.: listă circulară cu două metode: GetEvent și PutEvent

- Trimiterea și primirea de evenimente

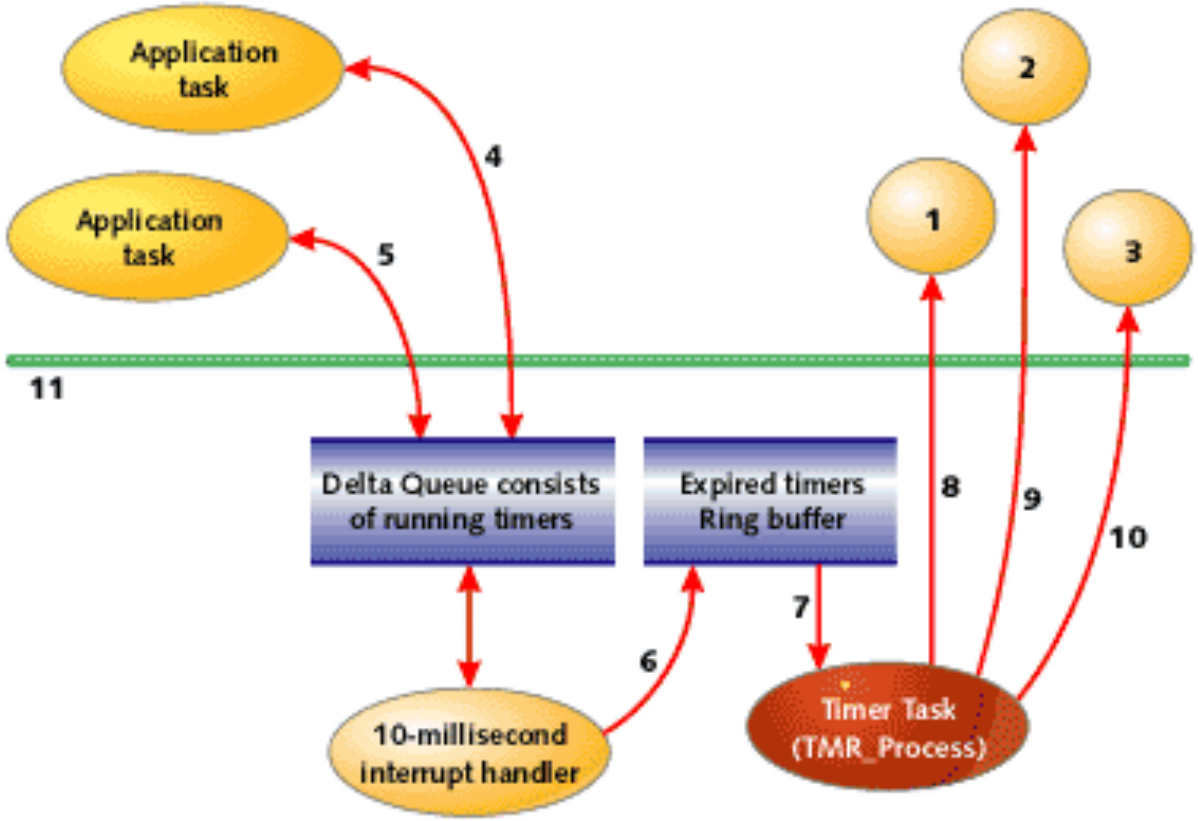
```
void IO_ProcessOutputs(void) {
   int ret;
   EVENT_TYPE OutputEvent;
   OutputEvent.NewState = 1;
   OutputEvent.Number = 1;
   PutEvent(&OutputEvent); // insereaza un eveniment in coada
   // .....
   if ((ret = GetEvent(&OutputEvent) != EMPTY) {
     // am primit un eveniment; proceseaza
     IO_OutputChange(OutputEvent.Number,OutputEvent.NewState)
   }
}
```

Implementarea operațiilor periodice simple

- Mai multe acțiuni trebuie executate cu exactitate sau periodic
  - Afișarea cursorului
  - O ieșire care trebuie închisă/deschisă periodic
  - Afișează un mesaj la o anumită oră sau periodic
  - Iluminează/stinge LCD-ul după un timp
- E mai dificil de implementat câte o funcție pentru fiecare
  - Se alocă mai multe variabile de incrementare în rutina de întrerupere a timerului
- O soluție: timer software

Timer Software

- Pentru fiecare timer folosit se definește o funcție care se execută la expirarea intervalului
- Rutine de TMR_Start si TMR_Stop
- Toate timerele din sistem sunt ținute în “coada delta” în ordinea expirării lor.
  - Timerele care expiră peste 10, 60, & 200 tick-uri de ceas vor fi stocate:
    - 10 50 (= 60 - 10) 140 (= 200 - 60)



Sisteme Foreground/Background

- Cea mai comun întalnită soluție hibridă pentru aplicațiile embedded simple
- Foloseste un fir de execuție interrupt-driven (foreground) ȘI un fir de execuție în bucla principala (background)
- Toate soluțiile real-time sunt niște cazuri speciale de sisteme foreground/background
- Polled loops = Sistem Background-only
- Sisteme Interrupt-only = Sisteme Foreground-only
- Tot ce nu este time-critical trebuie să fie în fundal
- Background este procesul cu prioritatea cea mai mică

Sisteme Foreground/Background

Foreground (interrupt) *(t1)*

```
on interrupt {
  do_clock_module();
  if(time_for_control) do_control();
}
```

Background *(t2)*

```
while (1) {
  if (time_for_display_update) do_display();
  else if (time_for_operator_input) do_operator();
  else if (time_for_request) do_request();
}
```

- Departajarea relaxează constrângerile: t1 + t2 ≤ 20 ms

Abordarea Multi-Tasking

- O buclă de control: un singur “task”
- Foreground/background: două task-uri
- Generalizare: task-uri multiple
  - Numite și procese, thread-uri
  - Fiecare proces se execută în paralel
    - Procesele interacționează simultan cu elementele externe
      - Monitorizează senzorii, controlează efectoarele, tratează , IO etc.
    - Creează iluzia paralelismului
  - Cerințe
    - Planificarea proceselor
    - Partajarea datelor între procese concurente

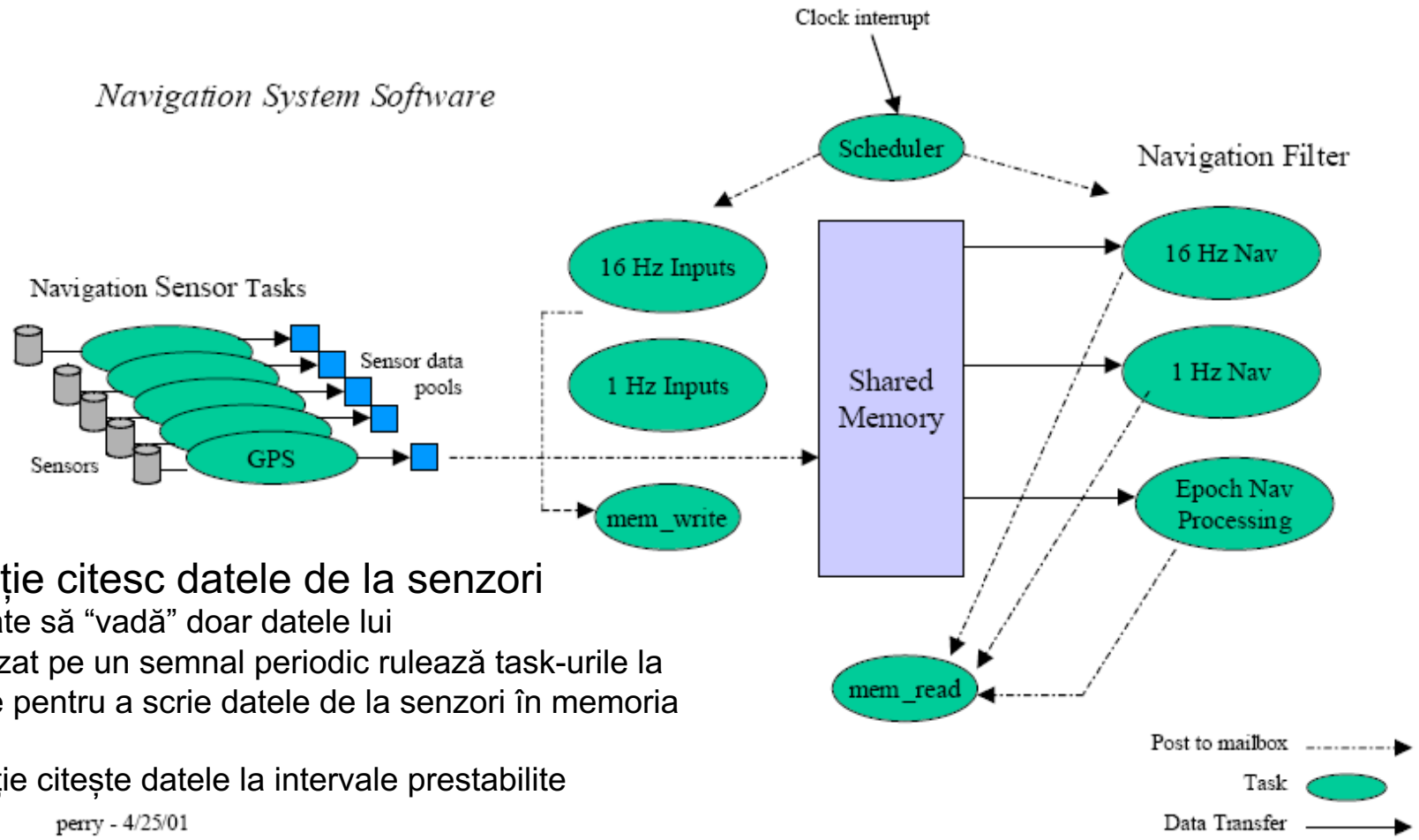
Multitasking

- Ce înseamnă Multitasking?
  - Procese separate împart același procesor (sau procesoare)
  - Fiecare task se execută în contextul propriu
  - Deține procesorul pentru cuanta curentă de timp
  - Are variabile proprii
  - Poate fi întrerupt
- Mai multe task-uri pot interacționa și funcționa ca un program unitar.

Caracteristicile unui task

- Un proces poate avea
  - Cerințe de resurse
  - Prioritate atașată
  - Relații de precedență
  - Cerințe de comunicare
  - Și, cel mai important, constrângeri de timp
    - » Specificarea momentului de timp la care să se execute sau să se termine o acțiune
    - » Ex. Perioada unui proces periodic
    - » Sau deadline pentru un proces neperiodic

Exemplu Multitasking



Task-urile de navigație citesc datele de la senzori

- Fiecare task poate să “vadă” doar datele lui
- Planificatorul bazat pe un semnal periodic rulează task-urile la frecvențe diferite pentru a scrie datele de la senzori în memoria partajată.
- Filtrul de navigație citește datele la intervale prestabilite

perry - 4/25/01
4

Multitasking

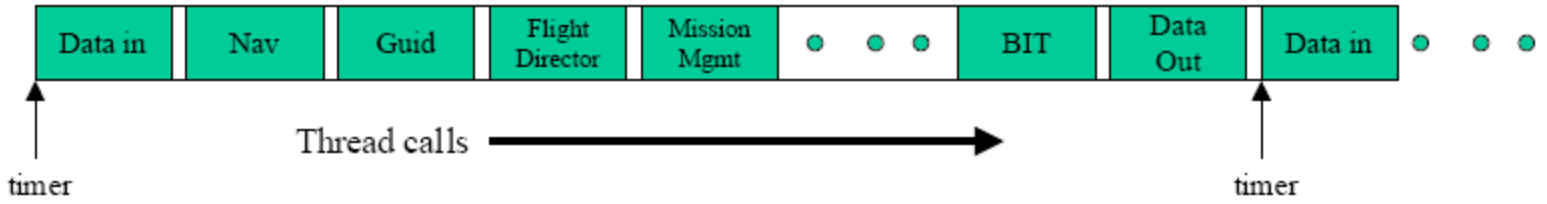
- Context switching
  - Atunci când CPU-ul schimbă un fir de execuție cu un altul se face o schimbare de context
  - Se salvează MINIMUL de informații menite să refacă procesul întrerupt la o apelare ulterioară
    - Exemplul cu cartea: De ce e nevoie? *(nume, pagina, paragraf , cuvantul_nr.)*
  - Într-un sistem de calcul MINIMUL este, de cele mai multe ori
    - Conținutul registrelor
    - Conținutul PC
    - Conținutul registrelor de coprocesor (dacă e cazul)
    - Adresa paginii de memorie
    - Adresele I/O-urilor mapate în memorie
    - Variabile speciale
  - În timpul schimbării contextului, întreruperile sunt dezactivate
  - Sistemele de timp real trebuie să aibă timpi minimi pentru context-switching.

Multitasking

- Câte task-uri împart același procesor?
  - Sisteme cu execuție ciclică
  - Sisteme round-robin
  - Sisteme preemptive

Sisteme cu execuție ciclică

- Folosește o planificare statică pentru a ordona toate firele de execuție



- Pro:
  - Ușor de implementat (folosite în sisteme critice și de menținere a vieții)
- Contra:
  - Nu sunt foarte eficiente d.p.d.v. al folosirii CPU
  - Nu permit un timp de răspuns optim (întotdeauna, unele sarcini au prioritate mai mare)

Sisteme Round-Robin

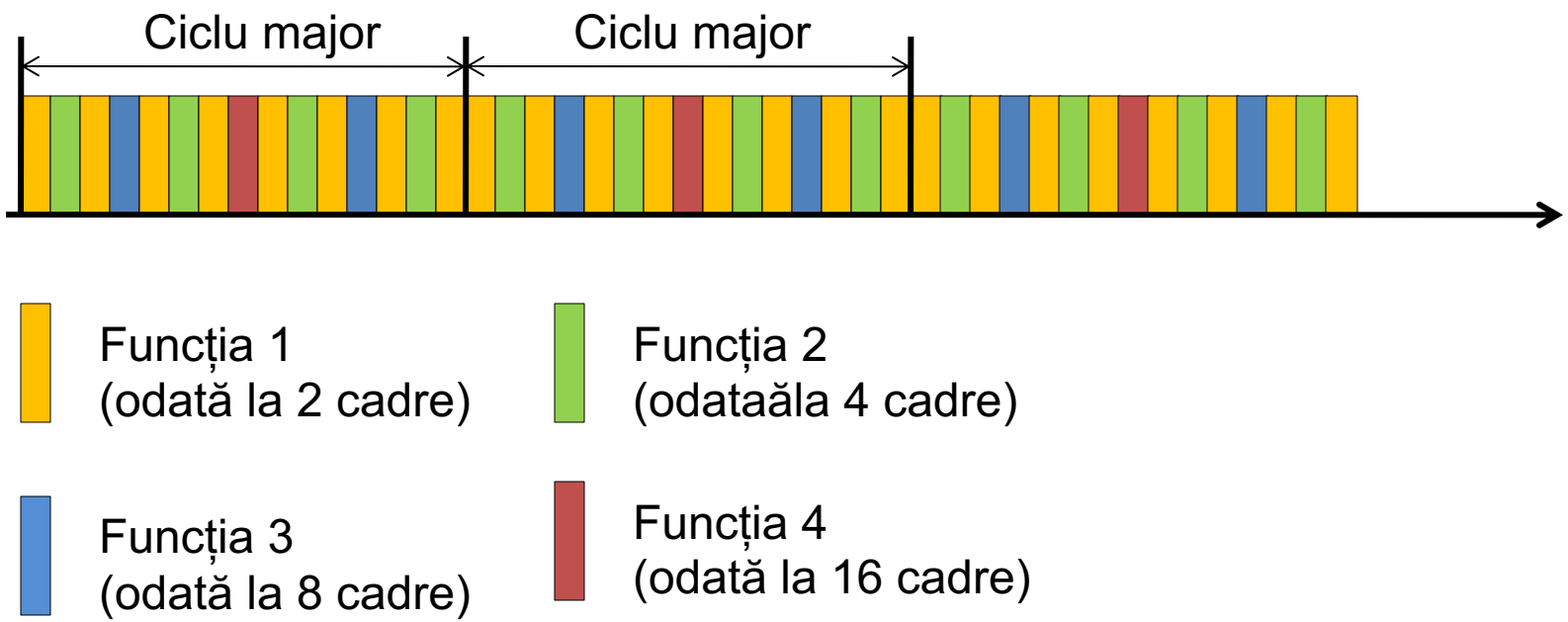
- Procesele se execută secvențial până la finalizarea tuturor
- De multe ori în conjuncție cu o schemă de execuție ciclică
- Fiecărui task îi este alocată o cuantă de timp.
- Există un timer de sistem care generează o întrerupere la expirarea fiecărei cuante de timp
- Task-ul se execută până la finalizare sau până la terminarea cuantei de timp alocate lui.
- Contextul este salvat sau refăcut la fiecare ieșire/intrare a procesului într-o cuantă de execuție.

Planificarea Statică

- Aplicabilă la task-urile periodice
- Se construiește o tabelă și fiecare proces are alocată o cuantă de timp
- Previzibil dar inflexibil
    - Tabela este total refacută când un singur proces își modifică timpul de execuție
- Timpul este împărțit în cicli minori (o cuantă) și un timer declanșează execuția procesului planificat pentru cuanta respectivă de timp.
- Un set de cicli minori constituie un ciclu major care se repetă countinuu.
- Operațiile sunt implementate ca niște proceduri și sunt incluse în liste de execuție pentru fiecare ciclu minor
- La începutul unui ciclu minor timerul apelează în ordine fiecare procedură din lista respectivă.
- Fără preemptare: operațiile lungi trebuie “sparte” pentru a putea începe într-un ciclu minor.

Exemplu

- Patru funcții care se execută la 50, 25, 12.5 si 6.25Hz (20,40,80 și 160ms) pot fi planificate într-o execuție ciclică cu un ciclu minor de 10ms:



Sisteme preemptive

- Un task cu prioritate mai mare poate să preempteze pe un al doilea, dacă acesta din urmă este în execuție în cuanta curentă de timp
- Prioritățile alocate fiecărui task sunt bazate pe urgența execuției fiecărui task în parte
- Prioritățile pot să fie fixe sau dinamice

Preemptare

- Non-preemptive: procesul, odată pornit nu se oprește decât după ce și-a terminat execuția
  - » Ex.: N task-uri, fiecare task j apelat la un interval T_j are nevoie de un timp C_j de execuție
    atunci: $T_j \geq C_1 + C_2 + \ldots + C_N$ în cel mai rău caz (toate celelalate N-1 task-uri sunt și ele gata)

- Preemptive: un proces poate fi oprit pentru rularea altui proces
  - Complică implementarea
  - Dar putem face mai bine planificare

Planificarea (scheduling)

- Date de intrare
  - Unul sau mai multe procese
  - Timpii de activare, execuție și deadline pentru fiecare proces

- Algoritm de planificare: politica de alocare a proceselor pe unul sau mai multe procesoare
- Planificare fezabilă: dacă algoritmul de planificare poate satisface toate constrângerile
- Algoritm optim: Un algoritm de planificare care produce un rezultat fezabil (dacă acesta există)

Evaluarea performanțelor algoritmilor de planificare

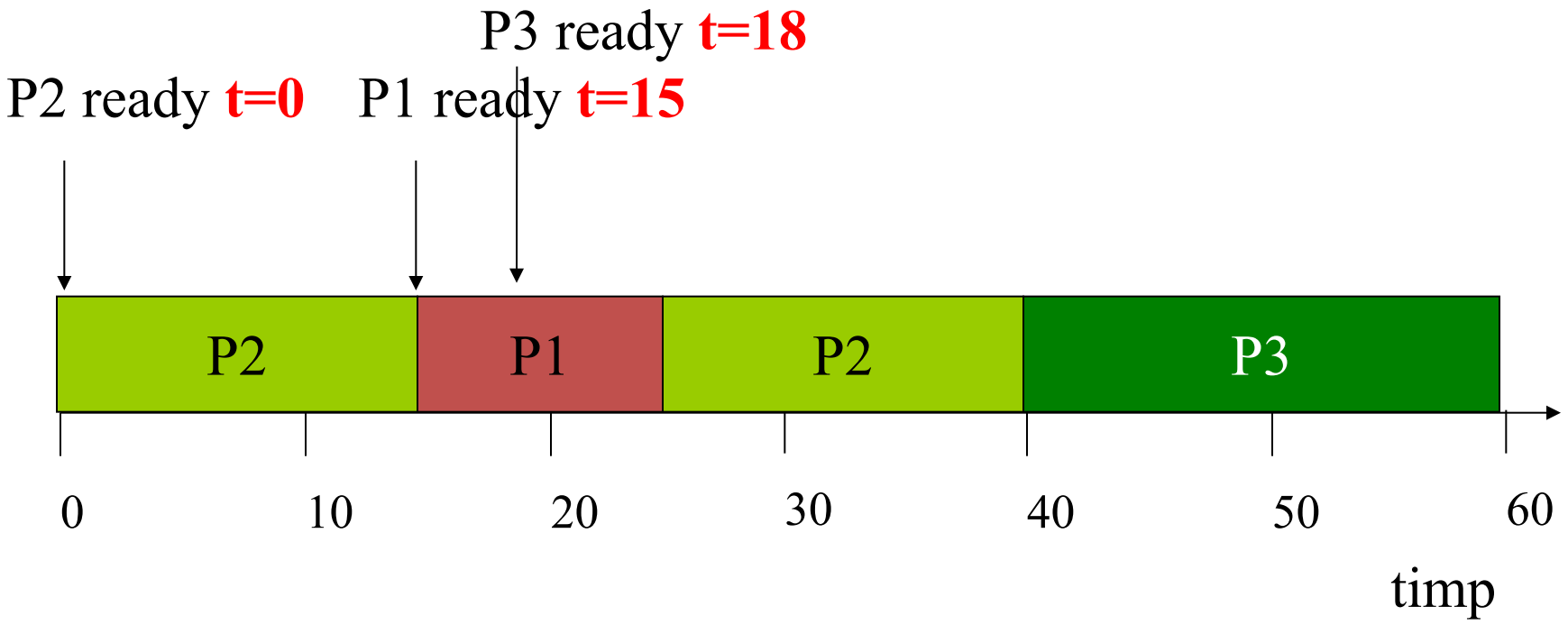
- Cazul static: planificare off-line care asigură că toate deadline-urile sunt satisfăcute
  - Metrică secundară:
    - » Maximizarea numărului mediu de sosiri devreme
    - » Minimizarea numărului mediu de întârzieri
- Cazul dinamic: nici o garanție a priori că termenul limită va fi satisfăcut
  - Metrică:
    - » Maximizarea numărului de procese care satisfac termenul limită
- Pentru amândouă cazurile:
  - Overhead de planificare minim

Planificarea bazată pe priorități

- Fiecărui proces i se atribuie o prioritate (static sau dinamic)
  - Atribuirea priorităților se face ținându-se cont de constrângerile de timp
  - Prioritatea statică: atrăgătoare pentru un sistem simplu (ieftin și nu necesită recalculări)
- La un moment de timp, rulează sarcina cu cea mai mare prioritate
  - Dacă rulează un proces cu prioritate mica și sosește un altul cu prioritate mai mare, primul proces este preemptat
- Atribuirea unor priorități corespunzătoare permite tratarea anumitor situații în timp real.

Exemplu

- P1: prioritate 1, timp execuție 10
- P2: prioritate 2, timp execuție 30
- P3: prioritate 3, timp execuție 20



Algoritmi de planificare pe prioritati

- Rate Montonic Scheduling (RMS)
  - Priorități statice bazate pe perioade de timp
    - » Prioritate mai mare pentru perioade mai scurte
  - Optim, între toți algoritmii de prioritizare statică
- Earliest-Deadline First (EDF)
  - Atribuire dinamică
  - Cu cât e mai aproape termenul unui proces, cu atât e mai mare prioritatea
  - Aplicabil atât pentru sarcinile periodice, cât și pentru cele neperiodice
  - Se recalculează prioritățile la sosirea unei sarcini noi
    - » Mai costisitor în privința overhead-ului obținut la rulare

Planificarea constrânsă de toleranța la defecte

- Exemplu: mecanism de impunere a unui termen limită pentru a garanta terminarea execuției unei sarcini principale dacă nu există nici un defect și alocarea unei perioade suplimentare de timp pentru intrarea în execuție a unei sarcini alternative (de precizie mai mică) în cazul unei avarii.
    - Dacă nu se produce avaria, timpul suplimentar alocat este reutilizat
- Altă abordare: scenariu de rezervă prevăzut în algoritmul de planificare normal și activat la producerea unei defecțiuni

Planificarea cu realocarea resurselor

- Sunt variații în timpii de execuție ai unui proces
  - Unele sarcini pot să termine mai devreme decât era planificat
- Dispecerul de sarcini poate să realoce acest timp pentru alte sarcini
  - Ex. Sarcinile care nu sunt de timp real pot fi rulate în intervalele realocate (idle slot)
  - Și mai bine: folosește timpul suplimentar pentru a garanta terminarea sarcinilor cu constrângeri de timp

Procesarea cu o anumită precizie

- Ideea principală: aranjează calculele a.î. să se producă rezultate mai precise dacă este mai mult timp la dispoziție
  - Dacă nu este suficient timp există totuși un rezultat de calitate mai proastă în loc de nimic
- Poate fi cuplată cu optimizarea consumului de energie (vezi cursurile anterioare)

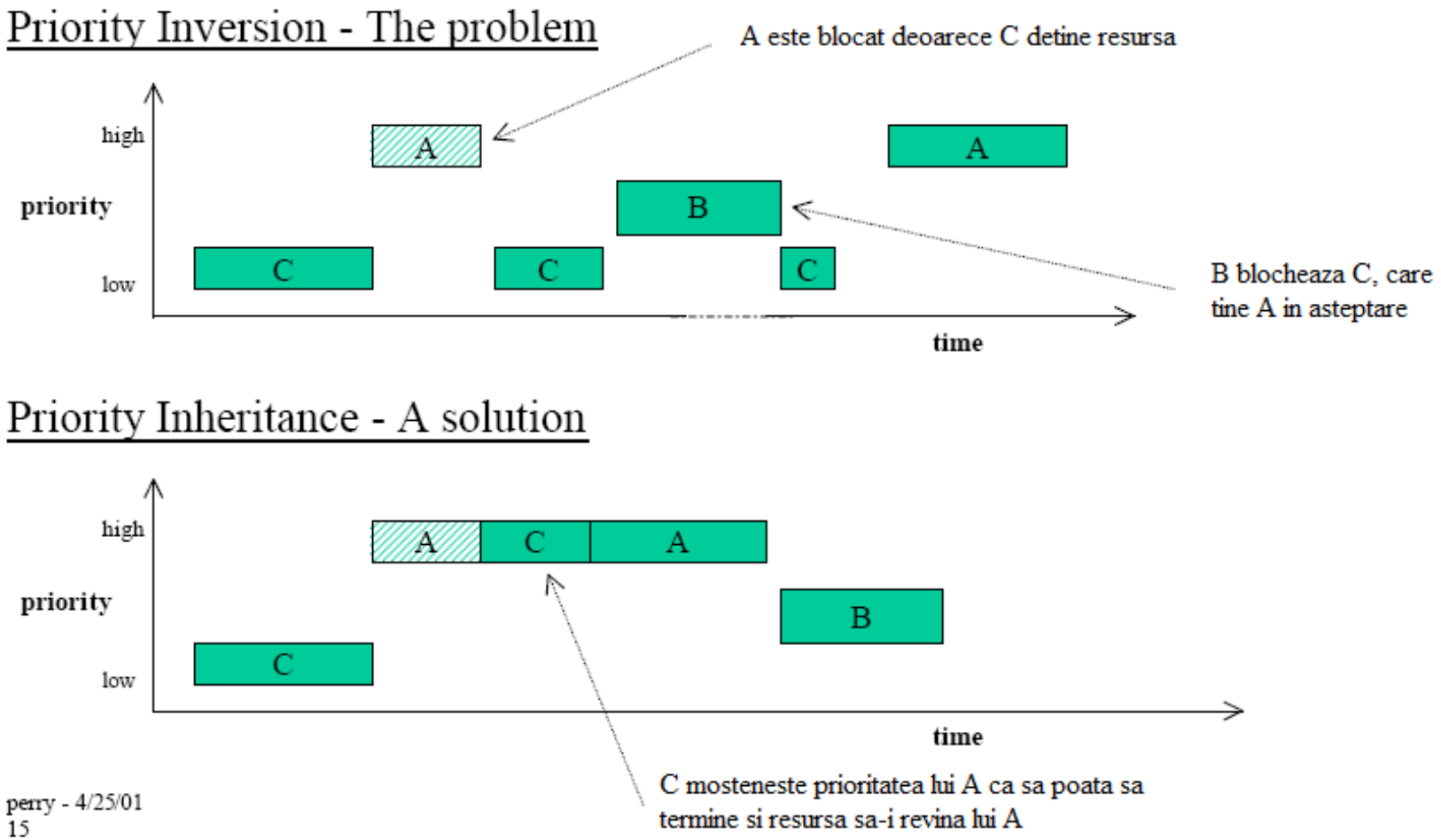
Multitasking

- Multitasking-ul nu este perfect
  - Task-urile de prioritate mare acaparează resursele și “înfometează” task-urile cu prioritate mică
  - Task-urile cu prioritate mică împart aceeași resursă cu cele de prioritate mare și le blochează pe acestea din urmă
- Cum tratează un RTOS aceste probleme?
  - Rate Monotonic Systems (frecvență de executie mare = prioritate mare)
  - Moștenirea priorității

Multitasking

- Priority Inversion / Priority Inheritance
  - Task A și Task C împart aceeași resursă
  - Task A este High Priority
  - Task C este Low Priority
  - Task A este blocat când Task C se execută (adică A ajunge să aibă prioritatea lui C - Priority Inversion)
  - Task A va fi blocat pentru și mai mult timp dacă Task B de prioritate medie vine și blochează Task C înainte ca acesta să termine
  - Un RTOS bun detectează această condiție și promovează temporar Task C la High Priority a Task A (Priority Inheritance)

Priority Inversion/Inheritance



RTOS Full-Feature

- Extind soluțiile foreground/background

  Adaugă:

  – Interfețe de rețea

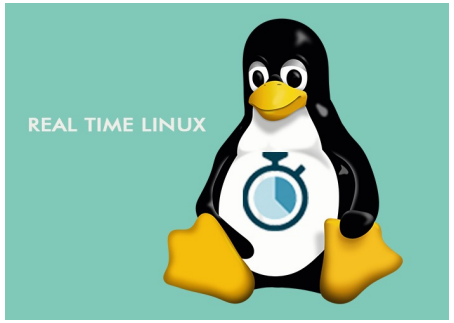
  – Drivere dispozitive

  – Soluții complexe pentru debugging

- Sunt alegerea cel mai des întâlnită pentru sistemele complexe
- Foarte multe sisteme de operare disponibile deja pe piață.

Câteva exemple de RTOS

- Pentru sisteme mici
  - NuttX
  - TinyOS (Berkeley)
  - ContikiOS
- Sisteme medii
  - FreeRTOS
- Sisteme complexe
  - VxWorks
  - Real-time Linux



TinyOS

- Kernel monolitic
- Scris în o variantă de C, nesC
- Algoritmi, protocoale, drivere, sisteme de fișiere și shell
- Dezvoltat pentru rețele de senzori
- Nu mai este dezvoltat activ

tinyos.net

ContikiOS

Embedded Systems Laboratory

- Arhitectura pe mai multe niveluri
- Scris în C, dar anumite părți pot fi scrise și în C++
- Servicii pentru device drivers, comunicație și manipularea datelor
- Stiva uIP, device driver loader și system Protothreading
- Protothreads sunt o implementare simplă de multithreading, fără a folosi stiva
- Code replacement la runtime

contiki-os.org/

Exemplu: proces periodic în ContikiOS

```
PROCESS_THREAD(etimer_process, ev, data) {
        static struct etimer et;
        PROCESS_BEGIN(); /* set timer to expire after 5 seconds */
        etimer_set(&et, CLOCK_SECOND * 5);
        while(1) {
                PROCESS_WAIT_EVENT(); /* Same thing as PROCESS_YIELD */
                if(etimer_expired(&et)) { /* Do the work here and restart timer to get it periodic !!! */
                        printf("etimer expired.\n");
                        etimer_restart(&et);
                }
        }
        PROCESS_END();
}
```

NuttX

Embedded Systems Laboratory

- Creat de Gregory E. Nutt
- Primul release public: feb 2007
- Rulează pe uC si uP de la 8 la 32 de biți
- BSD license
- Amprentă memorie foarte mică (zeci de kB)
- Foarte personalizabil
- Inspirat de Linux/Unix
  - VFS
  - MTD
  - PROCFS
  - NuttShell (nsh)

nuttx.apache.org

NuttX

- POSIX complaint
- Fully preemptible
- Virtual File System (VFS)
- Loadable kernel modules
- Symmetric Multi-Processing (SMP)
- Realtime scheduling (FIFO, RR, SPORADIC)
- Tickless operation support (lower power consumption)
- Pseudo-terminals (PTY) and I/O redirection

Real-time Linux & VxWorks

Embedded Systems Laboratory

- Microcontroller (MMU-less) :
  - uClinux - (kernel < 512KB) cu implementare full TCP/IP

- VxWorks
  - Multitasking
  - IPC – mutex, cozi de mesaje
  - Sistem de fisiere
  - IPV6
  - VxSim – simulator
  - Sisteme care folosesc VxWorks:
    - Boeing 787
    - ASIMO
    - Linksys WRT54G
    - F18 - Super Hornet
    - Mars Reconaissance Orbiter
    - Spirit & Opportunity Mars Exploration Rovers
    - Deep impact, Stardust

Alegerea unui RTOS

- De unde știu care este cea mai bună soluție pentru aplicația mea?
- Un indiciu important îl primesc de la modul cum ajung datele în sistemul meu
  - Neregulat (o secvență știută dar variabilă de intervale în care primesc date sau evenimente)
  - Rafală (secvență arbitrară limitată la un număr maxim de evenimente simultane)
  - Limitat (interval minim între două intrări succesive de date)
  - Limitat cu rata medie (intervale imprevizibile dar apropiate de o medie globală)
  - Nelimitat (pot să fac doar o predicție statistică)
- Care este I/O-ul critic?
- Există termene limită absolute?

Alegerea unui RTOS

- Exemple din viața reală:
  - Vehicul reutilizabil pentru lansarea sateliților pe orbită
    - Vectorul de control pentru propulsie necesită date care estimează altitudinea odată la 40msec sau racheta devine instabilă
  - *Execuție ciclică.*
  - Navigația și controlul unui submarin.
    - Interfață cu senzori multipli care sunt eșantionați cu rate diferite
    - Informația de la unitatea inerțială de referință este crucială dar temporizarea exactă a datelor de intrare nu este esențială
  - *Schemă de execuție preemptivă de pe un RTOS comercial. Task-urile importante au prioritatea cea mai mare.*

Alegerea unui RTOS

- Sistem de control avionică – are nevoie de date de la suprafețele de control al zborului și de la echipamentul de navigație la fiecare 50ms
  - *Execuție ciclică. Fiecare task rulează până la finalizare. Toate task-urile rulează în serie.*
  - *Ultimele task-uri s-ar putea să nu termine execuția până la apariția întreruperii de 50ms.*
- Microcontroller care operează antene radar și comută între ele pentru a determina poziția unui obiect. Dacă primește semnal, alimentează circuitul de procesare a semnalului.
  - *Polled loop.*

Concluzii

- Multi-tasking implică partajarea resurselor de către mai multe procese
- Fiecare task are contextul propriu de execuție
- Mai multe task-uri se pot combina pentru a forma un program
- Sunt mai multe căi de a implementa multi-taskingul
  - Cyclic Executive
  - Round Robin
  - Sisteme bazate pe priorități
- Unele sisteme sunt construite prin combinarea mai multor concepte RTOS
- Nu există o singură cale care sa fie garantat bună pentru implementarea unui sistem embedded dar există cu siguranță căi greșite. Alegeți-vă RTOS-ul potrivit aplicației!