

## 5. Multitasking si sisteme multiprocesor

### 5.1. Procese si fire de executie (thread)

### 5.2. Strategii de planificare a proceselor

### 5.3. Comunicatia inter-proces

### 5.4. Comunicatia in sistemele multiprocesor



ROMANIA EUROPEANA



MINISTERUL EDUCATIEI SI  
CERCETARII



FONDUL SOCIAL EUROPEAN  
2007-2013



INTERVENII STRUCTURALE  
2007-2013

### 5.1. Procese si fire de executie (thread)

#### 5.1.1. Notiuni fundamentale

**Multitasking** – este o metoda prin care procese multiple (tasks) gestioneaza in comun resurse de procesare (timp procesor – CPU). In cazul sistemelor cu un singur procesor un singur proces poate fi rulat la un anumit moment de timp. Pentru rularea *simultana* a mai multor procese, metoda multitasking alocă si programează timpii CPU de executie ai fiecarui proces astfel incat pentru un interval de timp acestea sa ruleze virtual simultan. Procesul de realocare a resurselor CPU de la un proces la altul (context switch) va trebui sa se desfasoare cu o viteza suficient de mare pentru ca acest paralelism sa poate fi fructificat. Chiar si in sistemele multiprocesor, in general, sunt rulate mai multe procese decat numarul de unitati de procesare (CPU).



### 5.1. Procese si fire de executie (thread)

#### 5.1.1. Notiuni fundamentale

**Sistemele de operare** pot sa utilizeze una din urmatoarele trei tehnici de rulare a proceselor in sisteme multitasking:

- *Multiprogramming* – procesul curent este mentinut in executie pana cand o operatie cu un periferic necesita introducerea unui timp de asteptare sau sistemul de operare decide intreruperea executiei procesului. Aceste sisteme maximizeaza utilizarea resurselor CPU;
- *Time-sharing* – procesul curent elibereaza resursele ocupate fie voluntar, fie la aparitia unui eveniment extern de tip intrerupere. Intreruperea voluntara va aparea la intervale fixe de timp generate de un timer intern;
- *Real-time* – sisteme in care prioritizarea proceselor corelata cu un sistem de intreruperi ierarhizat determina executia si tratarea *simultana* a evenimentelor externe.

### 5.1. Procese si fire de executie (thread)

#### 5.1.1. Notiuni fundamentale

##### Sisteme multiprogramming

Pentru primele sisteme de calcul costul timpului de procesare (CPU time) era ridicat si viteza perifericelor era scazuta. Astfel, de cele mai multe ori, accesarea unui periferic determina trecerea procesorului in stare de asteptare pana cand perifericul executa operatia curenta.

In 1960 s-au implementat primele sisteme multiprogram: cand programul curent ajungea la o operatie de citire a unui periferic, un alt program din lotul incarcat initial in memorie era rulat de CPU pana la finalizarea operatiei de accesare a perifericelor.

Dezavantaje:

- Variabilitatea timpului de executie al unui program

### 5.1. Procese si fire de executie (thread)

#### 5.1.1. Notiuni fundamentale

##### **Sisteme time-sharing – multitasking co-operativ**

Pentru primele sisteme time-sharing erau constituite din suite de aplicatii care isi transferau voluntar timpii de acces la CPU prin corelarea co-operanta a nevoilor de procesare.

Ca urmare a faptului ca sistemele de tip co-operativ cedeaza in mod regulat accesul la CPU, proiectarea atenta a acestor SO trebuie efectuata in scopul evitarii posibilitatilor de blocare, mai ales in medii aleatoare (retele).

Exemple: Windows (pre 9x), MacOS (pre Osx), Windows 9x 16 bit legacy apps.

### 5.1. Procese si fire de executie (thread)

#### 5.1.1. Notiuni fundamentale

##### **Sisteme time-sharing – multitasking pre-emptiv**

Permit sistemelor embedded sa asigure fiecarui proces o “felie” corespunzatoare din timpul CPU si sa gestioneze rapid o cantitate mare de evenimente externe ce trebuie corelate cu un anumit proces.

In fiecare moment specific de executie procesele sunt grupate in doua categorii:

- Procese care sunt legate de operatii I/O (I/O bound);
- Procese legate strict de CPU (CPU bound);

Sistemul de operare aloca (poll, busywait) timpi de executie catre CPU bound in timp ce asteapta o anumita intrerupere externa. Sosirea datelor I/O determina realocarea timpilor CPU catre I/O bound. Aceste sisteme au fost imbunatatite de aparitia co-procesoarelor.

Exemple: Sinclair, Windows 9x, NT, MacOSx, Unix.

Mixed pre-emptive and co-operative.



### 5.1. Procese si fire de executie (thread)

#### 5.1.1. Notiuni fundamentale

##### Sisteme real-time

Sisteme in care prioritizarea proceselor corelata cu un sistem de intreruperi ierarhizat determina executia si tratarea *simultana* a evenimentelor externe. Aceste sisteme permit definirea unor procese cheie a caror executie este prioritara.

Termenul (real-time) deriva din simularea proceselor si utilizarea lui implica o viteza de calcul suficient de mare ca sa poata sa urmareasca viteza de desfasurare a evenimentelor din realitate.

Un sistem va fi denumit *real-time* daca rezultatul proceselor rulate va fi nu numai corect ci si disponibil in timp util:

- Hard real-time – rezultatul este inutil dupa expirarea timpului (automotive systems);
- Soft real-time – este permisa existenta unor timpi de latentă in detrimentul calitatii (digital video broadcasting);

### 5.1. Procese si fire de executie (thread)

#### 5.1.1. Notiuni fundamentale

##### Threads – fire de executie

Firele de executie au aparut din ideea ca cel mai eficient mod de cooperare intre procese este prin gestionarea comuna a intregului spatiu de memorie. Astfel, firele de executie sunt procese care ruleaza in acelasi context de memorie, schimbarea executiei de la un thread la altul neafectand contextul memoriei utilizate de CPU.

Firele de executie sunt programate pre-emptiv ca urmare a gestionarii comune a resurselor.

Fibrele (fibres) sunt segmente de proces asemanatoare firelor de executie insa pot fi programate co-operativ.

Pentru sisteme cu un singur CPU, firele de executie pot fi rulate simultan prin multiplexarea timpilor de executie cu o viteza suficient de mare – scheduler.



ROMANIA EUROPEANA



MINISTERUL EDUCATIEI SI  
CERCETARII



FONDUL SOCIAL EUROPEAN  
2007-2013



INDICATORI STRUCTURALI  
2007-2013



### 5.1. Procese si fire de executie (thread)

#### 5.1.1. Notiuni fundamentale

##### Threads /procese

Diferente esentiale dintre aceste notiuni:

- Procesele sunt in general independente, firele sunt subseturi ale unui proces;
- Procesele contin informatii proprii de stare, firele au in comun aceleasi informatii de stare (memorie, resurse);
- Procesele au spatii de adresare separate, firele au acelasi spatiu de adresare;
- Procesele interactioneaza numai prin mecanisme proprii de inter-comunicare;
- Schimbarea contextului de executie intre fire este mai rapid decat intre procese.

Multithreading.



ROMANIA EUROPEANA



MINISTERUL EDUCATIEI SI  
CERCETARII



FONDUL SOCIAL EUROPEAN  
2007-2013



INTEGRAREA INFRUCTURALE  
2007-2013

### 5.2. Strategii de planificare a proceselor

#### 5.2.1. Sistemul de operare xOS

**xOS** = eXampleOS

Este un sistem de operare folosit pentru exemplificare si are caracteristicile oricarui OS real.

Sistemul de operare va decide care proces are acces la timpul CPU intr-un moment particular. In plus, va inregistra starea fiecarui proces (task's context). Pentru un procesor pe 16 biti, e.g. 80x86, acest context este reprezentat de continutul registrilor: CS si IP, SS and SP, Flags si DS, ES, SI, DI, AX, BX, CX si DX. Contextul va fi salvat intr-un bloc de control al proceselor. In C++, acest bloc va face parte din proces:

```
class Task
{
    public:
        Task(void (*function)(), Priority p, int
        stackSize);

        TaskId      id;
        Context      context;
        TaskState    state;
        Priority      priority;
        int *         pStack;
        Task *        pNext;

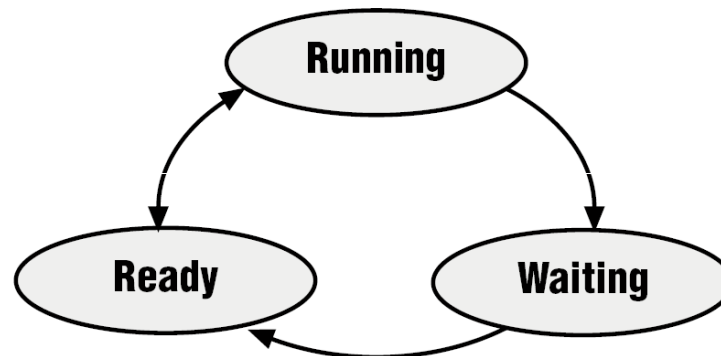
        void (*entryPoint)();

    private:
        static TaskId nextId;
};
```

### 5.2. Strategii de planificare a proceselor

#### 5.2.1. Sistemul de operare xOS

Orice proces poate sa se gaseasca la un moment dat intr-una din urmatoarele stari: *running*, *ready* si *waiting*.  
Un singur proces poate fi in starea *running* la un moment dat de timp si va iesi din aceasta stare numai la cererea sistemului de operare sau daca asteapta aparitia unui eveniment extern.



```
enum TaskState { Ready, Running, Waiting };
```

### 5.2. Strategii de planificare a proceselor

#### 5.2.2. Constructori

Pentru fiecare proces se va pune la dispozitie un constructor in care apelantul poate asigna functia, prioritatea si dimensiunea stivei pentru fiecare proces nou creat.

Primul parametru, *function*, este un pointer la functia care urmeaza sa fie executata in contextul noului proces.

Parametrul *p* este un numar unic (1,255) care reprezinta prioritatea noului proces si va fi utilizat de *scheduler* pentru selectarea urmatorului proces (255 are cea mai mare prioritate).

```
TaskId Task::nextId = 0;
/*****
* Method: Task()
* Description: Create a new task and initialize its state.
* Notes:
* Returns:
*****/
Task::Task(void (*function)(), Priority p, int stackSize)
{
    stackSize /= sizeof(int); // Convert bytes to words.
    enterCS(); // Critical Section Begin
    // Initialize the task-specific data.
    id = Task::nextId++;
    state = Ready;
    priority = p;
    entryPoint = function;
    pStack = new int[stackSize];
    pNext = NULL;
    // Initialize the processor context.
    contextInit(&context, run, this, pStack + stackSize);
    // Insert the task into the ready list.
    os.readyList.insert(this);
    os.schedule(); // Scheduling Point
    exitCS(); // Critical Section End
} /* Task() */
```

### 5.2. Strategii de planificare a proceselor

#### 5.2.2. Constructori

Partea functionala a constructorului este incadrata de doua apeluri de functii: *enterCS* si *exitCS*. Blocul incadrat de aceste apeluri este partea *critica* in care instructiunile trebuie executate *atomic* (in ordine si fara intreruperi).

*enterCS()* va salva starea intreruperilor si le va dezactiva iar *exitCS()* va reface contextul la iesire.

*contextInit()* – stabileste contextul procesului:  
pointer la structura de date a contextului,  
pointer la functia de start, pointer la noul proces construit, si pointer la noua stiva a procesului.

*os.readyList.insert(this)* – adauga procesul la lista de executie;

```
TaskId Task::nextId = 0;
/*****
* Method: Task()
* Description: Create a new task and initialize its state.
* Notes:
* Returns:
*****/
Task::Task(void (*function)(), Priority p, int stackSize)
{
    stackSize /= sizeof(int); // Convert bytes to words.
    enterCS(); // Critical Section Begin
    // Initialize the task-specific data.
    id = Task::nextId++;
    state = Ready;
    priority = p;
    entryPoint = function;
    pStack = new int[stackSize];
    pNext = NULL;
    // Initialize the processor context.
    contextInit(&context, run, this, pStack + stackSize);
    // Insert the task into the ready list.
    os.readyList.insert(this);
    os.schedule(); // Scheduling Point
    exitCS(); // Critical Section End
} /* Task() */
```

### 5.2. Strategii de planificare a proceselor

#### 5.2.3. Scheduler

Partea principala de planificare din SO. Planificarea executiei se face dupa algoritmi specifici.

1. **FIFO** – fiecare proces este rulat pana la finalizare (DOS). Un proces se poate autosuspenda, eliberand resursele ocupate, permitand comutarea de la un proces la altul (Win9x nu e multitasking).
2. **Shortest job first** – fiecare proces e rulat pana la finalizare (sau autosuspend) si urmatorul proces rulat e cel care solicita cel mai putin timp CPU.
3. **Round robin** – singurul algoritm in care procesele sunt pre-emptive (pot fi intrerupte in timpul rularii). In acest caz, fiecare proces poate rula doar un interval de timp predefinit, dupa care SO intrerupe procesul si da sansa altor procese sa ruleze. Procesul intrerupt nu are ocazia sa ruleze decat dupa ce fiecare proces din lista a primit timp CPU in runda respectiva.

### 5.2. Strategii de planificare a proceselor

#### 5.2.4. Embedded scheduler

Sistemele de operare embedded nu folosesc algoritmii simpli descrisi anterior. In sistemele embedded (in particular *real-time*) este necesara, de cele mai multe ori, gasirea unui algoritm care sa permita celui mai important proces sa ruleze in CPU in cel mai scurt timp. Astfel, in sistemele embedded este necesara utilizarea unui mecanism de prioritizare a proceselor care sa permita intreruperea procesului curent – trebuie garantat ca, in orice moment, procesul executat este procesul cu prioritatea cea mai mare care e gata sa fie executat.

Procese de prioritate scazuta trebuie sa astepte pana la finalizarea proceselor de prioritate mai mare. Pre-emptiv in acest caz inseamna ca orice proces poate fi intrerupt de SO daca un proces de prioritate mai mare devine pregatit sa ruleze (e in starea *ready*).

Conditiiile de intrerupere sunt seturi finite de instante de timp denumite puncte de planificare (*scheduling points*).

Pentru algoritmii de planificare prioritara trebuie pus la dispozitie si un mecanism de rezerva, pentru cazurile in care mai multe procese de aceeasi prioritate se gasesc deodata in starea *ready*. De obicei, mecanismul de rezerva este un algoritm simplu, de tipul *round robin*.

Se recomanda utilizatorilor de OS embedded sa utilizeze prioritati unice pentru fiecare proces creat pentru a evita situatiile descrise mai sus.

### 5.2. Strategii de planificare a proceselor

#### 5.2.4. Embedded scheduler

Clasa de implementare a planificarii este:

```
class Sched
{
    public:
        Sched();
        void start();
        void schedule();
        void enterIsr();
        void exitIsr();
        static Task * pRunningTask;
        static TaskList readyList;
        enum SchedState { Uninitialized, Initialized, Started };

    private:
        static SchedState state;
        static Task idleTask;
        static int interruptLevel;
        static int bSchedule;
};
```

Instantierea obiectuala a planificarii se va face:

```
extern Sched os;
```



### 5.2. Strategii de planificare a proceselor

#### 5.2.5. Puncte de planificare (scheduling points)

Punctele de planificare sunt un set de evenimente din SO in care *scheduler* este apelat (invocat). Doua astfel de evenimente au fost amintite la operatiile de creare/stergere a unor procese. In timpul acestor operatii, metoda *os.schedule* este apelata pentru a selecta urmatorul proces care trebuie rulat. Daca procesul curent are prioritate mai mare decat orice proces aflat in starea *ready*, va fi mentinut neintrerupt. In cazul stergerii proceselor aceasta obiectie va fi ignorata.

Un alt punct de planificare este generat de ceasul sistemului. Acest eveniment se genereaza utilizand un timer intern si este utilizat pentru intreruperea/oprirea unor procese care asteapta un timer software. In timpul acestui eveniment SO verifica starea tuturor timere-lor software si le decrementeaza, invoca *os.schedule* si verifica daca nu sunt procese noi, cu prioritate mare, trecute in starea *ready* de catre decrementarea timere-lor. Acest eveniment este, de obicei, programat la un interval cuprins intre 1ms.. 10ms.



ROMANIA EUROPEANA



MINISTERUL EDUCATIEI SI  
CERCETARII



FONDUL SOCIAL EUROPEAN  
2007-2013



INFRASTRUCTURA DE CERCETARE  
2007-2013

### 5.2. Strategii de planificare a proceselor

#### 5.2.5. Puncte de planificare (scheduling points)

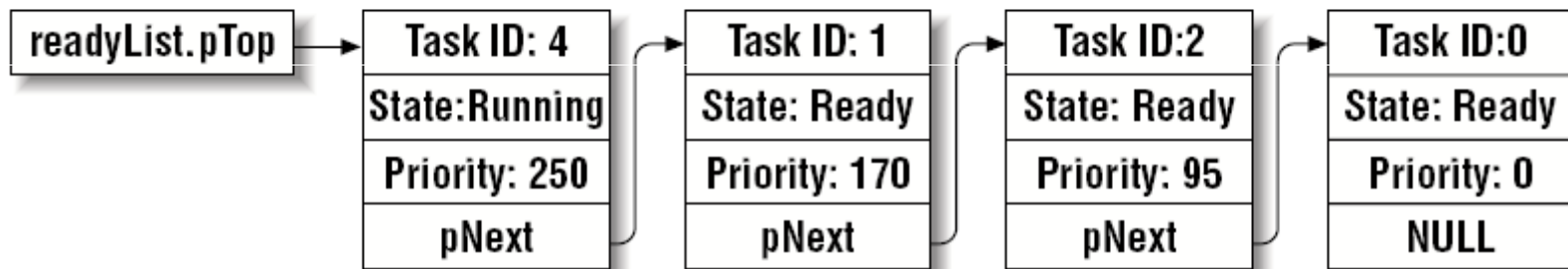
Punctele de planificare sunt un set de evenimente din SO in care *scheduler* este apelat (invocat). Doua astfel de evenimente au fost amintite la operatiile de creare/stergere a unor procese. In timpul acestor operatii, metoda *os.schedule* este apelata pentru a selecta urmatorul proces care trebuie rulat. Daca procesul curent are prioritate mai mare decat orice proces aflat in starea *ready*, va fi mentinut neintrerupt. In cazul stergerii proceselor aceasta obiectie va fi ignorata.

Un alt punct de planificare este generat de ceasul sistemului. Acest eveniment se genereaza utilizand un timer intern si este utilizat pentru intreruperea/oprirea unor procese care asteapta un timer software. In timpul acestui eveniment SO verifica starea tuturor timere-lor software si le decrementeaza, invoca *os.schedule* si verifica daca nu sunt procese noi, cu prioritate mare, trecute in starea *ready* de catre decrementarea timere-lor. Acest eveniment este, de obicei, programat la un interval cuprins intre 1ms.. 10ms.

### 5.2. Strategii de planificare a proceselor

#### 5.2.6. Lista de planificare (ready list)

Procedura *scheduler* utilizeaza o structura de date, lista de planificare, pentru urmarirea proceselor care se afla in starea *ready*. In xOS, aceasta structura se va implementa ca o lista simpla, ordonata dupa prioritatea proceselor. Astfel, primul element din lista va fi intotdeauna procesul cu prioritatea cea mai mare care e pregatit sa ruleze. In urma apelarii procedurii de planificare, acest element va trece in executie.



Avantaje: timpul de extractie minim – intotdeauna cel mai prioritar element va fi primul.

Dezavantaje: timpul de insertie mare – la fiecare insertie lista trebuie re-ordonata.

### 5.2. Strategii de planificare a proceselor

#### 5.2.7. Procesul *idle*

Daca in lista de procese nu exista niciun proces disponibil, apelarea procedurii *scheduler* va determina executia procesului *idle* – o bucla infinita care nu executa nimic. Acest proces este intotdeauna in starea *ready*.



ROMANIA EUROPEANA



MINISTERUL EDUCATIEI SI  
CERCETARII



FONDUL SOCIAL EUROPEAN  
2007-2013



INSTRUMENTE STRUCTURALE  
2007-2013

### 5.2. Strategii de planificare a proceselor

#### 5.2.8. Exemplu de *scheduler*

```
/* *****  
 * Method: schedule()  
 * Description: Select a new task to be run.  
 * Notes: If this routine is called from within an ISR, the  
 * schedule will be postponed until the nesting level  
 * returns to zero.  
 * The caller is responsible for disabling interrupts.  
 * Returns: None defined.  
 * *****/  
void  
Sched::schedule(void)  
{  
    Task * pOldTask;  
    Task * pNewTask;  
    if (state != Started) return;  
    // Postpone rescheduling until all interrupts are completed.  
    if (interruptLevel != 0)  
    {  
        bSchedule = 1;  
        return;  
    }  
    // If there is a higher-priority ready task, switch to it.  
    if (pRunningTask != readyList.pTop)  
    {  
        pOldTask = pRunningTask;  
        pNewTask = readyList.pTop;  
        pNewTask->state = Running;  
        pRunningTask = pNewTask;  
        if (pOldTask == NULL)  
        {  
            contextSwitch(NULL, &pNewTask->context);  
        }  
    }  
    else  
    {  
        pOldTask->state = Ready;  
        contextSwitch(&pOldTask->context, &pNewTask->context);  
    }  
    } /* schedule() */
```

### 5.2. Strategii de planificare a proceselor

#### 5.2.8. Exemplu de *scheduler*

Exista doua situatii in care procedura *scheduler* nu va determina o comutare a contextului:

1. Cand optiunea *multitasking* nu este activata – in general programatorii vor vrea sa creeze procesele inainte de activarea procedurii *scheduler*. In acest caz, rutina *main* a aplicatiei va arata ca mai jos – de fiecare data cand un proces este creat procedura *scheduler* este activata insa, ca urmare a faptului ca aceasta verifica intai valoarea variabilei *state* (daca optiunea *multitasking* e activa), nu va face schimbarea contextului pana la apelarea rutinei *start*.

```
#include "xOS.h"
void taskAfunction(void);
void taskBfunction(void);
/*Create 2 tasks, each with its own unique function and priority */
    Task taskA(taskAfunction, 150, 256);
    Task taskB(taskBfunction, 200, 256);
/*****
* Function: main()
* Description: This is what an application program might look like
* if ADEOS were used as the operating system. This
* function is responsible for starting the operating
* system only.
* Notes: Any code placed after the call to os.start() will
* never be executed. This is because main() is not a
* task, so it does not get a chance to run once the
* scheduler is started.
* Returns: This function will never return!
*****/
void
main(void)
{
    os.start();
    // This point will never be reached.
} /* main() */
```

### 5.2. Strategii de planificare a proceselor

#### 5.2.8. Exemplu de *scheduler*

```
#include "xOS.h"
void taskAfunction(void);
void taskBfunction(void);
/*Create 2 tasks, each with its own unique function and priority */
    Task taskA(taskAfunction, 150, 256);
    Task taskB(taskBfunction, 200, 256);
/*****
* Function: main()
* Description: This is what an application program might look like
* if ADEOS were used as the operating system. This
* function is responsible for starting the operating
* system only.
void
main(void)
{
    os.start();
    // This point will never be reached.
}/* main() */
* Notes: Any code placed after the call to os.start() will
* never be executed. This is because main() is not a
* task, so it does not get a chance to run once the
* scheduler is started.
* Returns: This function will never return!
*****/
```

2. In timpul unei intreruperi, deoarece OS va verifica de fiecare data nivelul intreruperii curente si nu va permite schimbarea contextului numai in cazul nivelului zero. Daca procedura *scheduler* este apelata de un serviciu ISR (ca in cazul timerului), flagul *bSchedule* va indica faptul ca *scheduler* trebuie apelat la iesirea din subrutina de tratare a ISR. Acest proces permite sistemelor embedded sa aiba un raspuns rapid la intreruperi.

### 5.3. Comunicatia interproces

#### 5.3.1. Comutarea contextului

Schimbarea executiei de la un proces la altul = comutarea contextului. Deoarece acest context este specific fiecarui procesor, rutina software de comutare este scrisa, de obicei, in limbaj de asamblare. Pentru exemplificare s-a folosit pseudocod:

```
void  
contextSwitch(PContext pOldContext, PContext pNewContext)  
{  
    if (saveContext(pOldContext))  
    {  
        //  
        // Restore new context only on a nonzero exit from saveContext().  
        //  
        restoreContext(pNewContext);  
        // This line is never executed!  
    }  
    // Instead, the restored task continues to execute at this point.  
}
```



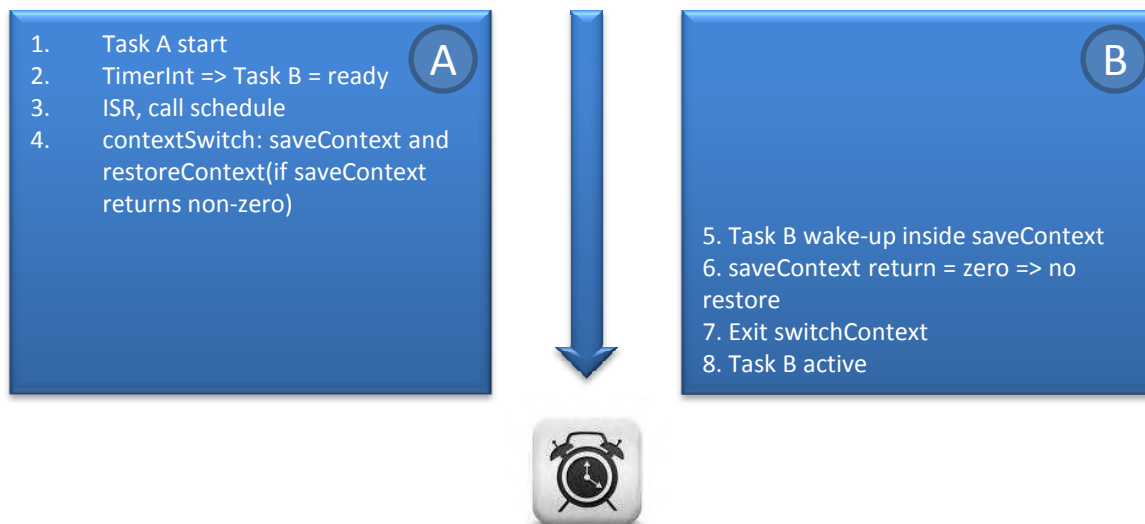
### 5.3. Comunicatia interproces

#### 5.3.1. Comutarea contextului

Procedura *contextSwitch* este apelata de catre *scheduler*, care este la randul sau apelat dintr-o zona a sistemului de operare in care intreruperile au fost deja dezactivate.

La comutarea contextului, cand noul proces intra in executie, vechiul proces trebuie sa fie trecut in starea *ready*.

Rutina *saveContext* executa salvarea contextului de executie si returneaza catre rutina de comutare a contextului o valoare nonzero atunci cand procesul este eliberat din executie si o valoare zero cand procesul este activat. Aceasta valoare este utilizata de *contextSwitch* pentru a decide daca trebuie apelata rutina *restoreContext*.



### 5.3. Comunicatia interproces

#### 5.3.2. Sincronizarea proceselor

Deși ne referim în mod frecvent la procese (în cadrul sistemelor multitasking) ca fiind entități independente, această imagine este una simplificată; de cele mai multe ori procesele trebuie să comunice între ele pentru a îndeplini funcționalitatea algoritmilor și trebuie, astfel, sincronizate. În cazul proceselor care folosesc același buffer de date, structura care realizează sincronizarea este denumită *mutex* (*mutual exclusion*).

Rolul componentei *mutex* este de a proteja resursele partajate (variabile globale, buffer-e de memorie, registrii) care sunt accesate de mai multe procese. Un *mutex* va limita accesul la astfel de resurse la un proces la un anumit moment de timp. Partile software care accesează resursele partajate conțin secțiuni critice de cod. Dacă în SO pentru aceste secțiuni s-au dezactivat întreruperile, pentru cazul în care procesele rulează acest lucru nu este de dorit. O structură *mutex* va proteja resursele partajate fără a fi nevoie să dezactivăm întreruperile.

### 5.3. Comunicatia interproces

#### 5.3.3. Componenta mutex

Definirea componentei *mutex* pentru xOS este detaliata mai jos:

```
class Mutex
{
    public:
        Mutex();
        void take(void);
        void release(void);
    private:
        TaskList waitingList;
        enum { Available, Held } state;
};
```

### 5.3. Comunicatia interproces

#### 5.3.3. Componenta mutex

Constructorul componentei *mutex* poate fi cel din exemplul urmator:

```
/******  
*  
* Method: Mutex()  
*  
* Description: Create a new mutex.  
*  
* Notes:  
*  
* Returns:  
*  
*****/  
Mutex::Mutex()  
{  
    enterCS(); // Critical Section Begin  
    state = Available;  
    waitingList.pTop = NULL;  
    exitCS(); // Critical Section End  
} /* Mutex() */
```

### 5.3. Comunicatia interproces

#### 5.3.3. Componenta mutex

Toate componentele *mutex* create in starea *Available* sunt asociat cu o lista inlantuita de procese in asteptare (care initial este nepopulata). Metoda de schimbare a starii *mutex* este metoda *take*, apelata de procese inainte de a folosi o resursa partajata. Cand apelarea acestei metode returneaza un rezultat, procesului apelant I se garanteaza accesul exclusiv la resursa partajata. Rutina *take* este descrisa mai jos:

```
/* *****  
 * Method: take()  
 * Desc.: Wait for a mutex to become available, then take it.  
 * Notes:  
 * Returns: None defined.  
 *****  
 */  
void  
Mutex::take(void)  
{  
    Task * pCallingTask;  
    enterCS(); // Critical Section Begin  
    if (state == Available)  
    {  
        //  
        // The mutex is available. Simply take it and return.  
        //  
        state = Held;  
        waitingList.pTop = NULL;  
    }  
    else  
    {  
        //  
        // The mutex is taken. Add the calling task to the waiting  
        list.  
        //  
        pCallingTask = os.pRunningTask;  
        pCallingTask->state = Waiting;  
        os.readyList.remove(pCallingTask);  
        waitingList.insert(pCallingTask);  
        os.schedule(); // Scheduling Point  
        // When the mutex is released, the caller begins executing  
        here.  
    }  
    exitCS(); // Critical Section End  
} /* take() */
```

### 5.3. Comunicatia interproces

#### 5.3.3. Componenta mutex

Daca structura *mutex* este utilizata de un alt proces (flag binar setat), procesul apelant va fi suspendat pana cand structura *mutex* este eliberata. Este posibil ca mai multe procese sa fie trecute in asteptare pentru acelasi *mutex*, lista asociat va fi ordonat in functie de prioritatea proceselor.

Eliberarea structurii *mutex* se va face cu metoda *release*, preferabil de catre procesul care a utilizat metoda *take*. Este posibil ca utilizarea metodei *release* sa activeze un proces de prioritate mai mare (care astepta in lista) ceea ce sa duca la suspendarea procesului curent.

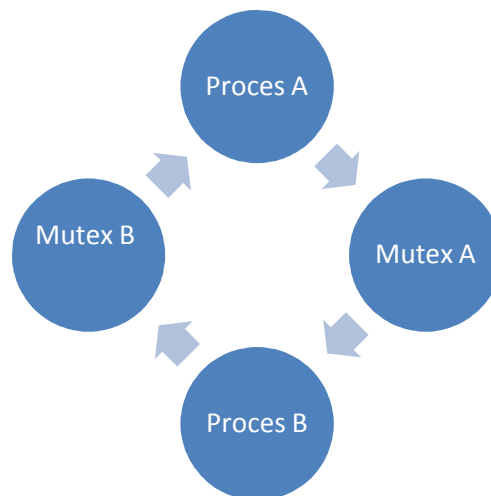
```
/******  
*** Method: release()  
* Desc.: Release a mutex that is held by the calling task.  
* Notes:  
*Returns: None defined.  
*****/  
void  
Mutex::release(void)  
{  
    Task * pWaitingTask;  
    enterCS(); // Critical Section Begins  
    if (state == Held)  
    {  
        pWaitingTask = waitingList.pTop;  
        if (pWaitingTask != NULL)  
        {  
            // Wake the first task on the waiting list.  
            waitingList.pTop = pWaitingTask->pNext;  
            pWaitingTask->state = Ready;  
            os.readyList.insert(pWaitingTask);  
            os.schedule(); // Scheduling Point  
        }  
        else  
        {  
            state = Available;  
        }  
    }  
    exitCS(); // Critical Section End  
} /* release() */
```

### 5.3. Comunicatia interproces

#### 5.3.4. Blocarea circulara

Structurile *mutex* reprezinta un instrument puternic de sincronizare a proceselor atunci cand partajeaza resurse comune. Doua dintre cele mai mari probleme ale acestor structuri sunt, insa, blocarea circulara si inversia de prioritate.

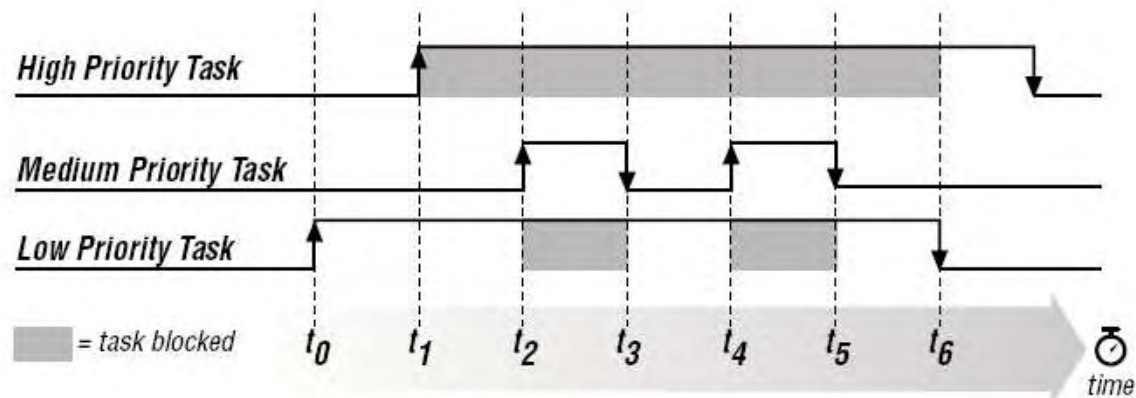
**Blocarea circulara** apare ori de cate ori exista o dependenta circulara intre procese si resurse. Sa presupunem ca avem doua procese, fiecare dintre ele utilizand cate o structura *mutex* proprie: A si B. Daca un proces ocupa *mutex* A si asteapta eliberarea structurii *mutex* B, in timp ce celalalt proces ocupa *mutex* B si asteapta eliberarea *mutex* A, atunci ambele procese sunt suspendate in asteptarea unui eveniment imposibil. Solutie: reboot.



### 5.3. Comunicatia interproces

#### 5.3.5. Inversia de prioritate

**Inversia de prioritate** apare ori de cate ori un proces de prioritate mare este suspendat in asteptarea unei structuri *mutex* care este ocupata de un proces de prioritate scazuta. Aparent, aceasta nu pare sa fie o problema, din moment ce procesele sunt astfel proiectate incat sa ia in considerare faptul ca resursele partajate ar putea fi ocupate in anumite momente. Situatia se schimba atunci cand apare un al treilea proces, care are o prioritate intermediara, intre nivelele celor doua initiale. Sa presupunem ca procesul cel mai putin prioritar este primul activat si ocupa structura *mutex*. Cand procesul cel mai prioritar devine *ready*, va fi blocat pana cand structura *mutex* este eliberata. Problema apare cand se activeaza procesul de prioritate medie (devine *ready*) si intrerupe procesul de prioritate mica (are prioritate mai mare – chiar daca nu are nevoie de *mutex*) ajungand sa ruleze in procesor, intarziind rularea procesului de mare prioritate. Solutie: imprumutul de prioritate – prioritatea procesului de mica prioritate este crescuta pana la nivelul procesului cel mai prioritar care are nevoie de *mutex*.





### 5.3. Comunicatia interproces

#### 5.3.6. Caracteristicile sistemelor *real-time* - RTOS

In inginerie, termenul *real-time* este folosit pentru a descrie probleme pentru care raspunsul intarziat este un raspuns fals. Aceste procese au termen de executie si sistemele embedded trebuie sa se conformeze acestor constrangeri.

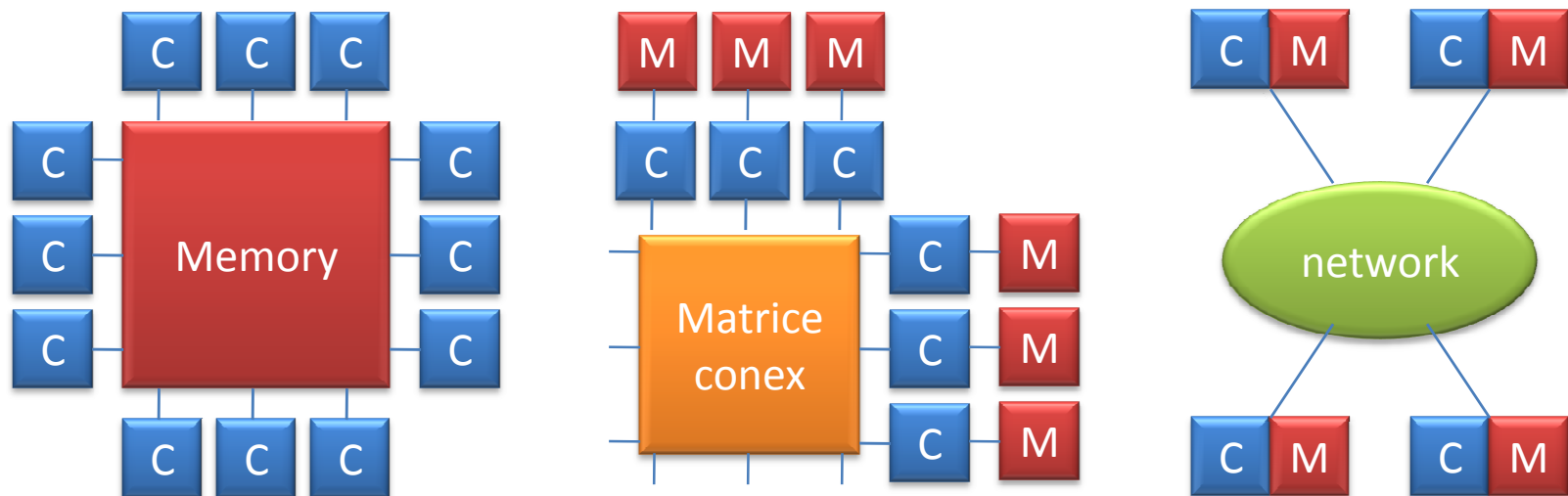
1. Pentru a califica un sistem de operare ca fiind RTOS, acesta trebuie sa fie deterministic si sa aiba timpi garantati de comutare a contextului pentru cazul cel mai defavorabil. Un astfel de sistem este deterministic daca toate apelurile sistem sunt calculabile in cazul cel mai defavorabil.
2. Latenta intreruperii = timpul total necesar de la frontul intreruperii pana la executia ISR => sectiunile critice, in care intreruperile sunt dezactivate, trebuie reduse si gasite alte cai de protectie a acestor sectiuni. Un timp de raspuns rezonabil = 1us..100us (in functie de aplicatie).
3. Timpul necesar la comutarea contextului – ar trebui sa fie mult mai mic decat timpul necesar rularii celui mai scurt proces (activitatea CPU sa fie orientata spre proces nu spre SO).

### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.1. Sisteme multiprocesor

**Definitie:** Sunt sisteme cu unitati de executie CPU multiple.

**Simetrie:** intr-un sistem multiprocesor toate unitatile CPU cu aceeasi functie trebuie sa fie identice cu exceptia celor cu functie speciala. Proiectarea aplicatiilor SW pentru aceste sisteme trebuie sa ia in considerare aceasta simetrie (sau lipsa ei) – de ex. Raspunsul la intreruperile HW se va gestiona de un singur CPU, sau pot fi exceptii SW (kerneluri) care trebuie rulate explicit.



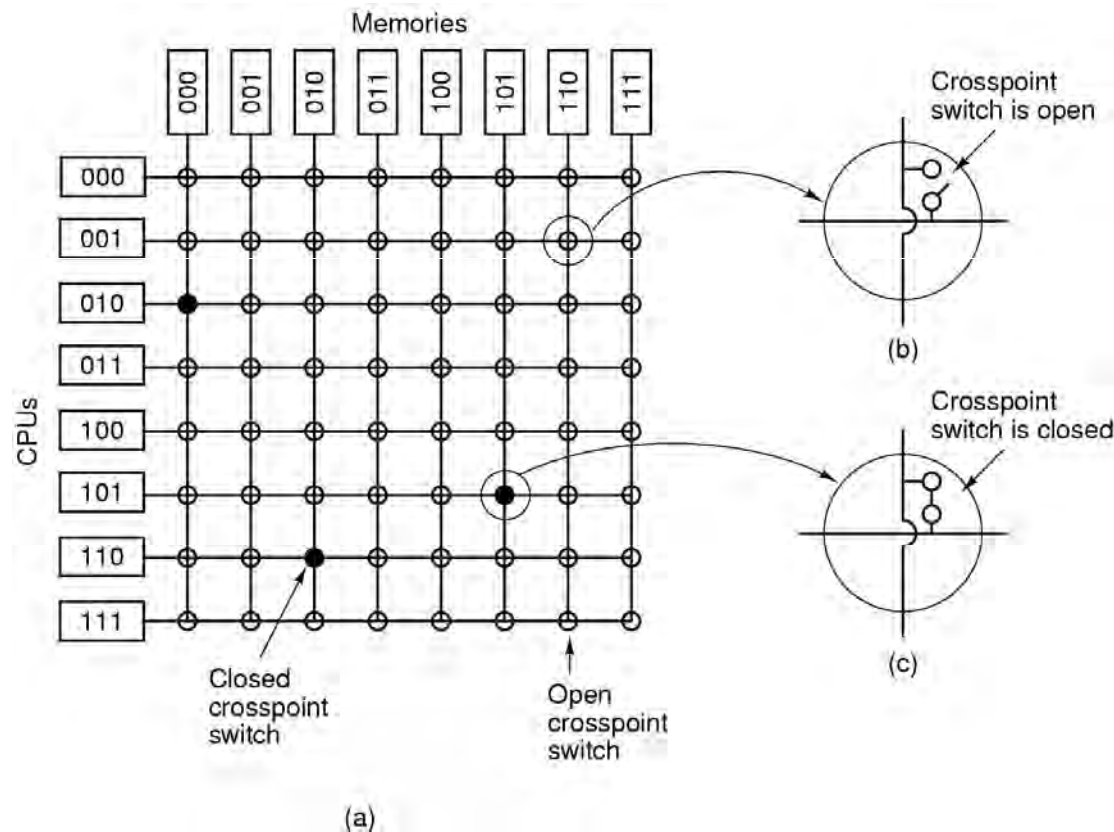


### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.1. Sisteme multiprocesor

##### Clasificare:

2. UMA – sisteme cu acces neuniform la memorie ;



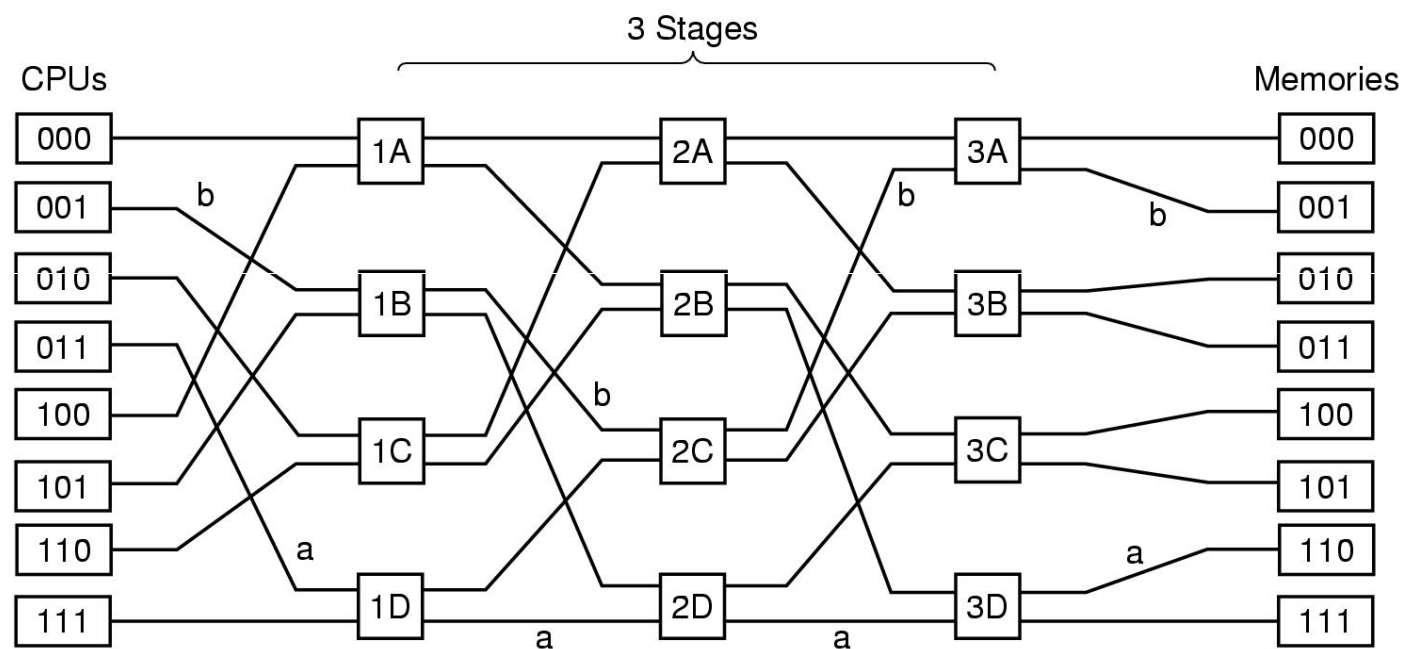
Sistem multiprocesor UMA  
cu matrice de conectare a  
memoriilor – non blocking  
network.

### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.1. Sisteme multiprocesor

##### Clasificare:

2. UMA – sisteme cu acces neuniform la memorie;



Sistem multiprocesor UMA  
cu retea multistage  
(Omega) de conectare a  
memoriilor – blocking  
network.

## 5. Multitasking si sisteme multiprocesor

### 5.4. Comunicatia in sistemele multiprocesor

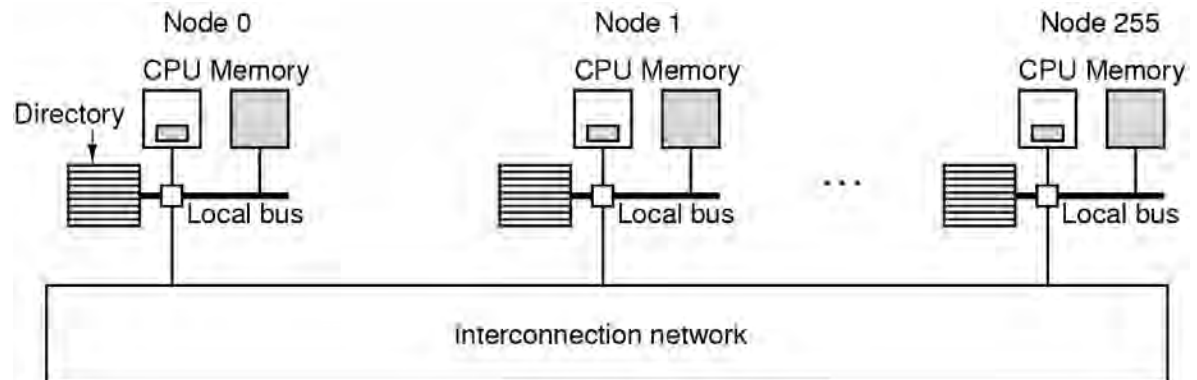
#### 5.4.1. Sisteme multiprocesor

##### Clasificare:

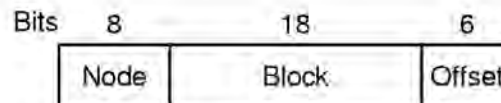
2. NUMA – sisteme cu acces neuniform la memorie;

##### Caracteristici:

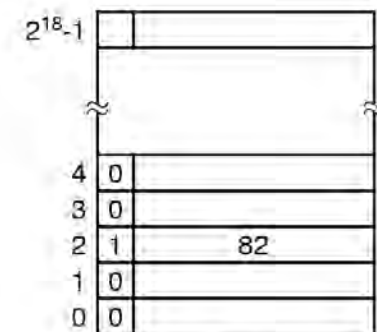
- Spatiu de adresare a memoriei unic si vizibil tuturor unitatilor CPU;
- Acces la memorie de tip “remote” utilizand comenzi LOAD si SAVE;
- Acces remote mai lent decat cel la memoria locala;



(a)



(b)



(c)

(a) Sistem multiprocesor cu 256

noduri “directory” CPU

(b) Campurile adreselor de memorie  
pe 32-biti;

(c) Directory at node 36

### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.1. Sisteme multiprocesor

##### Procesarea instructiunilor si a datelor:

1. SIMD – single instruction multiple data: o singura secventa de instructiuni in contexte diferite - vector;
2. MISD – multiple instruction single data: mai multe secvente in acelasi context – redundante, pipeline, hyper-threading;
3. MIMD – multiple instruction multiple data;

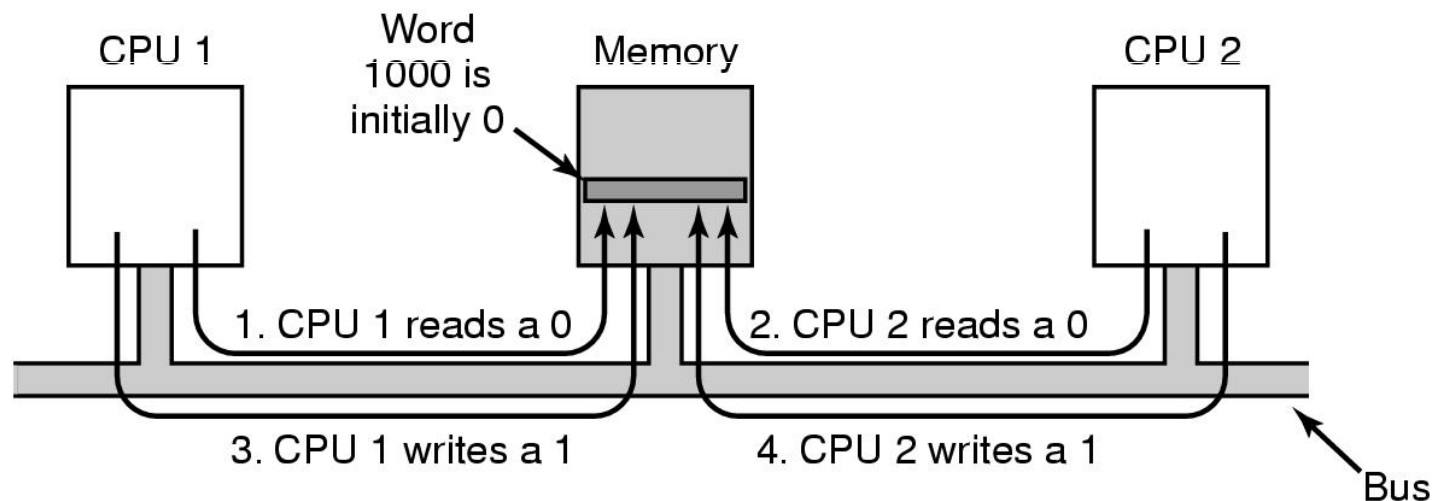
**Cuplarea procesoarelor:** bus-level sau cluster-level

### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.2. Sincronizarea sistemelor multiprocesor

##### Mutex si cache

In cazul sistemelor uniprocessor zonele critice de executie sunt protejate cu structuri mutex si cu optiuni de dezactivare a intreruperilor. In cazul sistemelor multiprocesor, dezactivarea intreruperilor functioneaza doar pentru procesorul activ, alte CPU pot continua sa ruleze si sa afecteze zone comune de memorie in acest timp. Astfel, metoda cea mai comuna de sincronizare este aceea de a proiecta structuri mutex de exclusiune mutuala (TSL – test and set lock).





### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.2. Sincronizarea sistemelor multiprocesor

##### **Bus lock**

In cazul sistemelor uniprocessor instructiunile TSL trebuie sa blocheze intai magistrala de acces, sa acceseze locatia de memorie si apoi sa deblocheze bus-ul. Acest protocol se poate implementa doar pe bu-surile care au linie dedicata pentru blocare. Pentru bus-urile fara aceasta linie dedicata, se poate implementa protocolul Peterson.

##### **Spin lock**

Utilizarea mutex TSL ca mai sus presupune trecerea procesorului exclus in starea de spin lock in care executa continuu o bucla scurta de testare a deblocarii bus-ului. Acest lucru nu numai ca iroseste timpul procesorului dar poate incarca masiv bus-ul memoriei incetinind functionarea altor CPU. Se recomanda algoritmul cu blocari multiple.



ROMANIA EUROPEANA



MINISTERUL EDUCATIEI SI  
CERCETARII



FONDUL SOCIAL EUROPEAN  
2007-2013

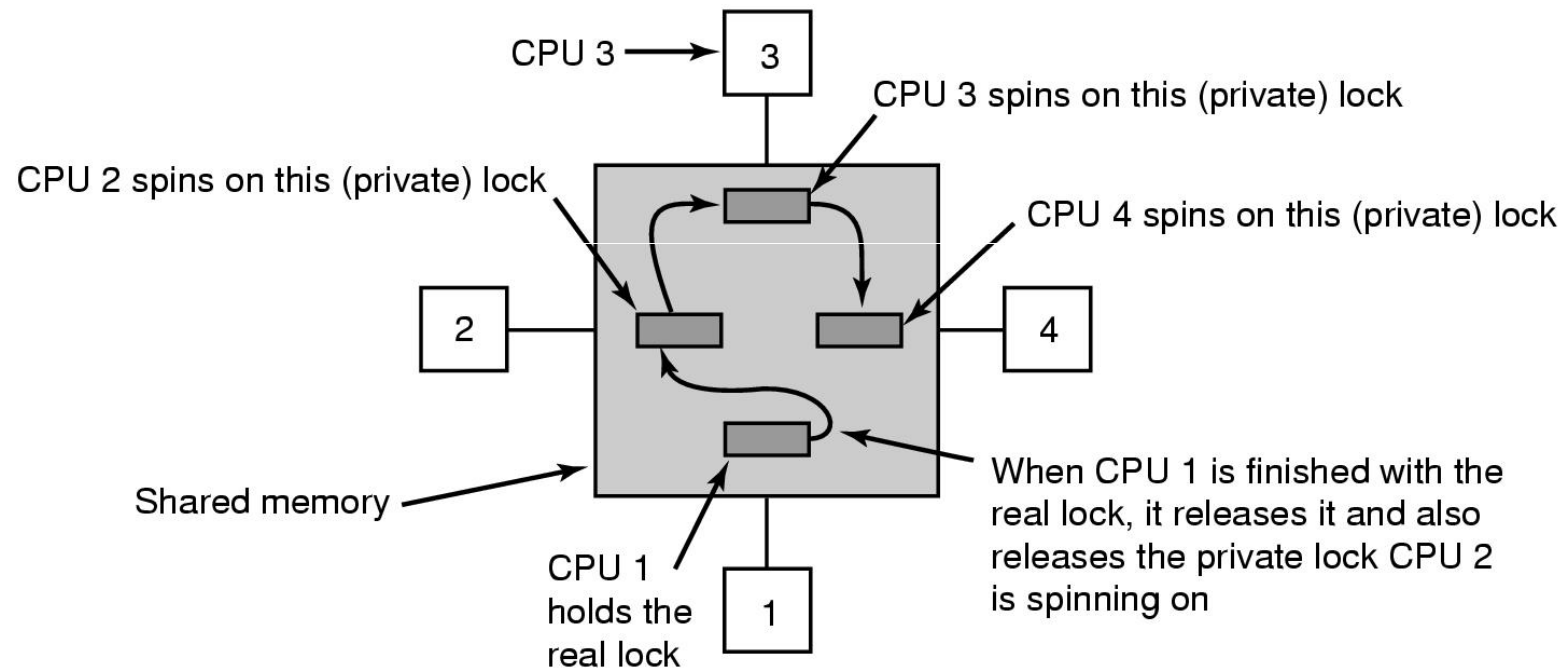


INFRASTRUCTURA DE  
CERCETARE SI DEZVOLTARE  
2007-2013

### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.2. Sincronizarea sistemelor multiprocesor

##### Multiple locks



### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.2. Sincronizarea sistemelor multiprocesor

##### Spinning si switching

Pana acum am presupus ca un CPU care are nevoie de un mutex blocat va astepta deblocarea lui fie prin testare continua (polling), fie prin testare intermitenta, fie prin atasarea lui in lista de CPU aflate in asteptare (bus timing list). In unele cazuri (cand ceea ce urmeaza sa execute e blocat in mutex) procesorul nu poate decat sa astepte. In alte cazuri insa poate sa schimbe procesul curent (switching) in loc sa astepte (spinning). Presupunand ca atat spinning cat si switching sunt disponibile pentru CPU trebuie analizata oportunitatea deciziei:

- Spinning-ul iroseste timp CPU direct – testarea continua a mutexului e un proces neproductiv;
- Switching – el necesita si el timp CPU din moment ce procesul curent trebuie salvat si trebuie repornit altul. Mai mult cache-ul procesorului va fi inutil si el trebuie reincarcat. Revenirea din switching va consuma din nou timp CPU.
- Presupunand ca timpul de blocare al unui mutex este de 50ms si ca este necesara 1ms pentru un switch si inca 1ms pentru revenirea din switch atunci switching-ul e de preferat. In cazul blocarii mutex-ului pentru 10ms, spinning-ul e de preferat.

In unele sisteme proiectantii prefera spinning, in altele switching. Doar analiza retrospectiva poate sa arate eficienta sistemului. In alte sisteme, CPU este pus in spinning pentru o durata de timp (threshold = minim timpul de switching) si apoi face switching daca mutex-ul nu este eliberat. Treshhold-ul poate fi static sau dinamic, in functie de istoricul timpilor de eliberare ai mutex-ului.

Cele mai bune rezultate se obtin atunci cand sistemul pastreaza istoricul ultimelor spin-uri si ia decizia in functie de acestea.

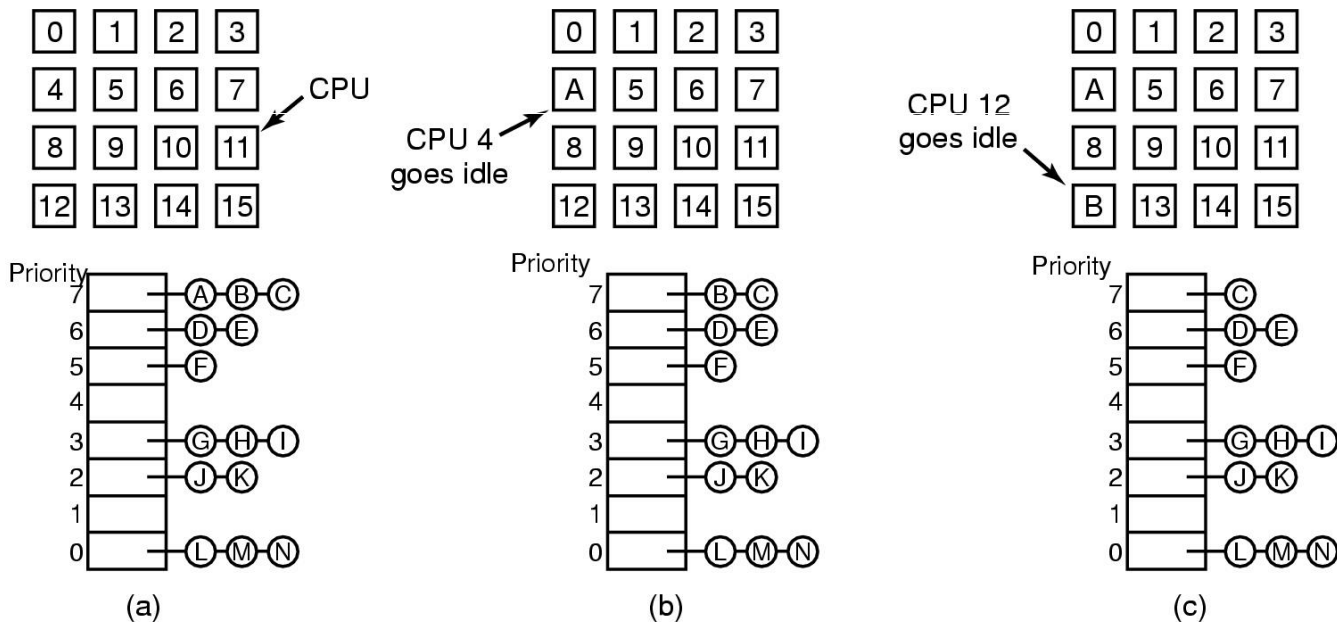
### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.3. Planificarea sistemelor multiprocesor

##### Multiprocessor scheduler

Acesta trebuie sa indeplineasca urmatoarele roluri:

- Sa asigneze procese catre CPU;
- Sa realizeze multiprogramarea la nivel de CPU;
- Sa arbitreze accesul;



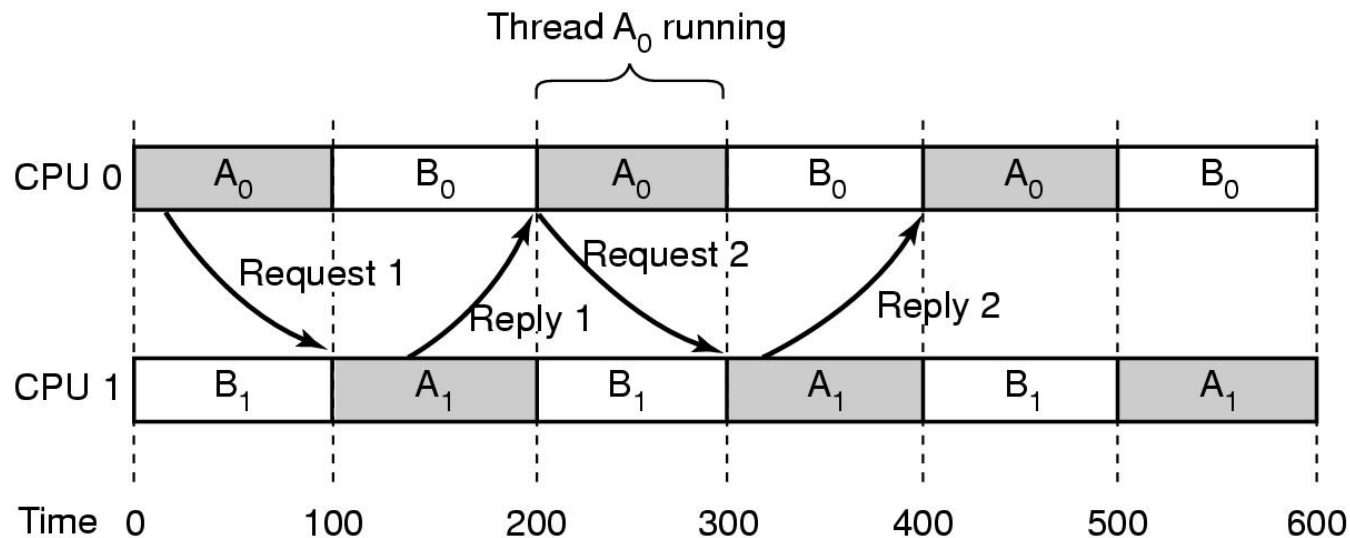
### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.3. Planificarea sistemelor multiprocesor

##### Multiprocessor scheduler

Trei tipuri de alocare a proceselor intre CPU:

1. Load sharing – partajarea incarcarii = procesele nu sunt alocate specific fiecarui procesor ci se urmareste o curba de incarcare uniforma;
2. Gang scheduling – un set de procese similare sunt grupate in seturi alocate unui (unor) CPU;
3. Alocare dedicata – procese specifice sunt alocate unor procesoare specifice;



### 5.4. Comunicatia in sistemele multiprocesor

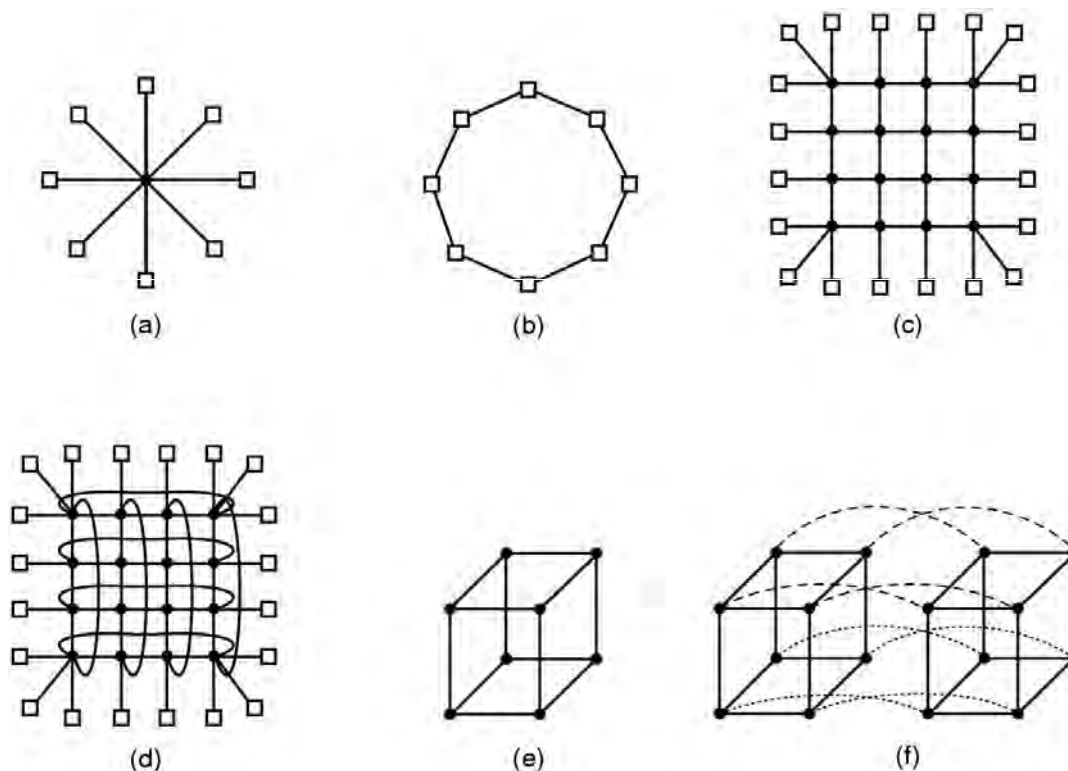
#### 5.4.4. Sisteme multicomputer – cluster computers, cluster of workstations (COW's)

Alternativa la symmetric multiprocessing (SMP)

Grup de unitati de calcul (computere) interconectate care lucreaza impreuna ca o singura unitate.

Topologii de interconectare

- (a) single switch
- (b) ring
- (c) Grid
- (d) double torus
- (e) cube
- (f) hypercube

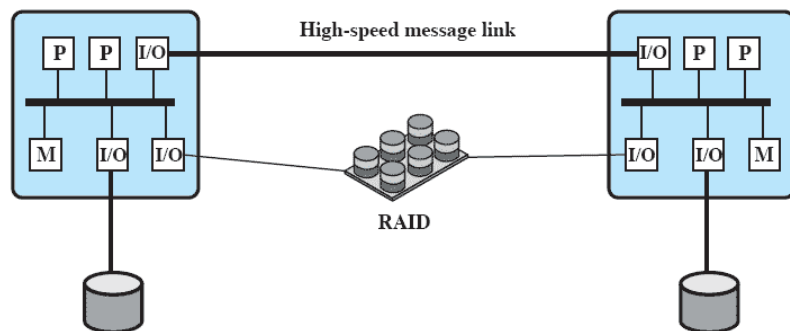


### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.4. Sisteme multicomputer – cluster computers, cluster of workstations (COW's)



(a) Standby server with no shared disk



(b) Shared disk

Clustering Method	Description	Benefits	Limitations
<b>Passive Standby</b>	A secondary server takes over in case of primary server failure.	Easy to implement.	High cost because the secondary server is unavailable for other processing tasks.
<b>Active Secondary</b>	The secondary server is also used for processing tasks.	Reduced cost because secondary servers can be used for processing.	Increased complexity.
<b>Separate Servers</b>	Separate servers have their own disks. Data is continuously copied from primary to secondary server.	High availability.	High network and server overhead due to copying operations.
<b>Servers Connected to Disks</b>	Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server.	Reduced network and server overhead due to elimination of copying operations.	Usually requires disk mirroring or RAID technology to compensate for risk of disk failure.
<b>Servers Share Disks</b>	Multiple servers simultaneously share access to disks.	Low network and server overhead. Reduced risk of downtime caused by disk failure.	Requires lock manager software. Usually used with disk mirroring or RAID technology.

### 5.4. Comunicatia in sistemele multiprocesor

#### 5.4.5. Sisteme multiprocesor: SMP vs. COW

##### **Avantaje SMP:**

- SMP sunt mai usor de configurat si de gestionat
- SMP ocupa mai putin spatiu si necesita mai putina energie
- SMP sunt produse stabile si documentate

##### **Avantaje COW:**

- COW ofera scalabilitate mai mare
- COW ofera disponibilitate mai mare



ROMANIA EUROPEANA



MINISTERUL EDUCATIEI SI  
CERCETARII



FONDUL SOCIAL EUROPEAN  
2007-2013



INDICATORI STRUCTURALI  
2007-2013