

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.  
}
```

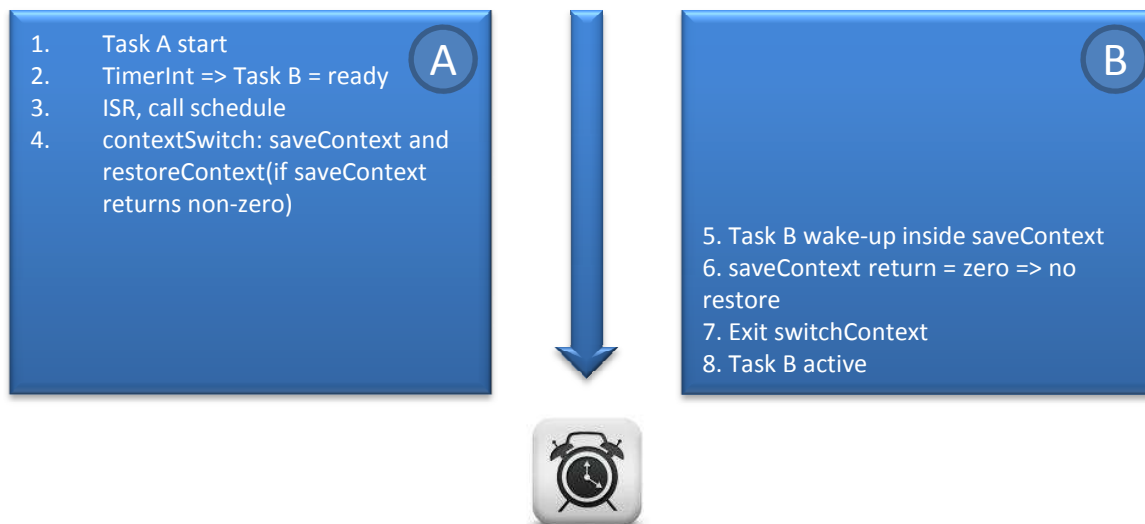
5.3. Comunicatia interproces

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.

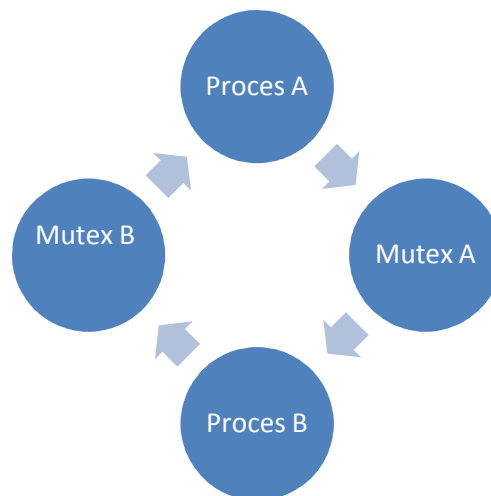
```
/******  
*** Method: release()  
* Desc.: Release a mutex that is held by the calling task.  
* Notes:  
*Returns: None defined.  
*****/  
void  
Mutex::release(void)  
{  
    Task * pWaitingTask;  
    enterCS(); // Critical Section Begins  
    if (state == Held)  
    {  
        pWaitingTask = 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.

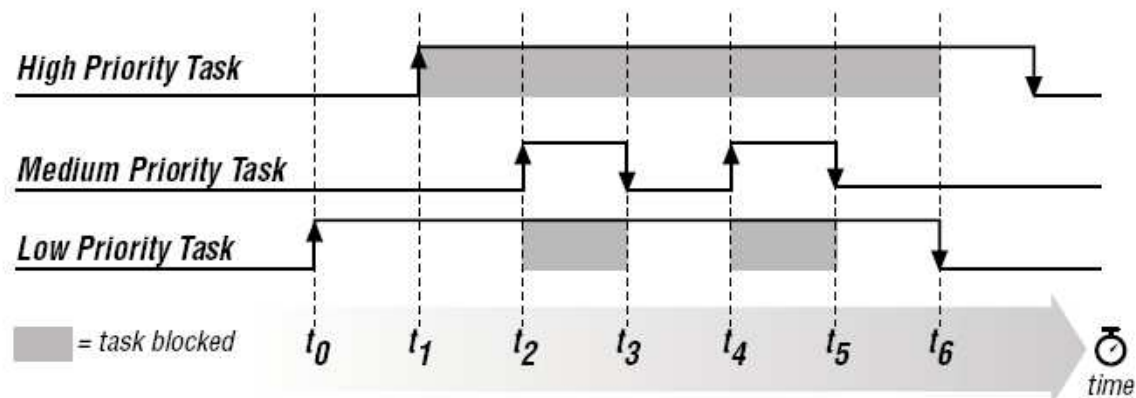
Blocarea circulara apare ori de cate ori exista o dependenta circulara intre procese si resurse. Sa presupunem ca avem doua procese, fiecare dintre ele utilizand cate o structura *mutex* proprie: A si B. Daca un proces ocupa *mutex* A si asteapta eliberarea structurii *mutex* B, in timp ce celalalt proces 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).