

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;

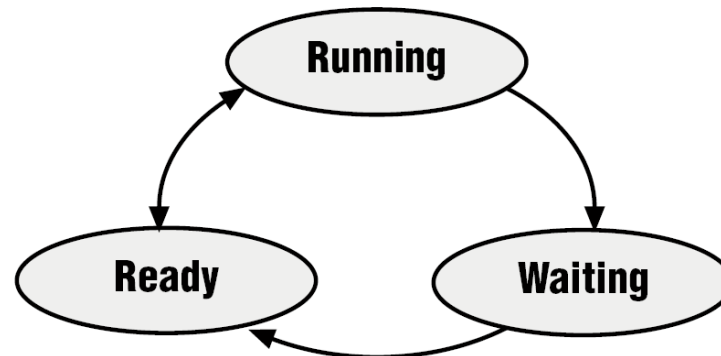
        void (*entryPoint)();

    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 rprioritatea 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*).

Condiitiile 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 rzerwa, 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.

5.2. Strategii de planificare a proceselor

5.2.5. Puncte de planificare (scheduling points)

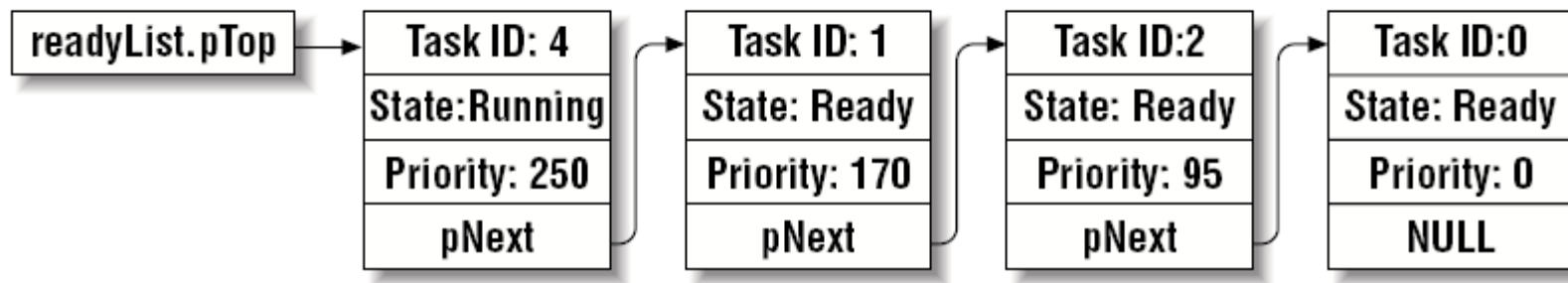
Punctele de planificare sunt un set de evenimente din SO in care *scheduler* este apelat (invocat). Doua astfel de evenimente au fost amintite la operatiile de creare/stergere a unor procese. In timpul acestor operatii, metoda *os.schedule* este apelata pentru a selecta urmatorul proces care trebuie 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*.



UNIA EUROPEANĂ



MINISTERUL EDUCAȚIEI ȘI
CERCETĂRII ȘTIINȚIFICE



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.
* 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() */
```

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.