SEMNALE (continuare)

Semnale din partea sistemului de operare (continuare)

SIGBUS

- emis de sistemul de operare in urma unei hotărîri a utilizatorului și reprezintă o eroare de magistrală
- nu se primește pe toate sistemele de operare (ci in functie de hardware, de ex. pe procesoarele Intel nu are loc)

Avem, de exemplu, instrucțiunile:

```
MOV EAX, addr;    //descarcă un registru pe 32 de biți la o adresă
MOV AX, addr;     //descarcă un registru pe 16 de biți la o adresă
MOV AL, addr;     //descarcă un registru pe 8 de biți la o adresă
```

Pe procesoarele Intel durata execuției celei de-a doua instrucțiuni diferă în funcție de valoarea *addr*:

- dacă este pară, se execută într-un timp;
- dacă este impară, se execută în doi timoi;

Similar, avem pentru prima instrucțiune:

- dacă este multiplu de 4, se execută într-un timp;
- dacă este pară, dar nu multiplu de 4, se execută în doi timpi;
- dacă este impară se execută în 4 timpi;

OBSERVAȚIE:

Compilatoarele așază variabilele la cea mai „bună” adresă posibilă;
Nu avem control asupra alegerii adresei la care se așază în memorie.

ex.: int i,j; // nu putem face o afirmație privind modul în care este așezat j față de i
int t[2]; // variabilele tabloului sunt așezate una după alta

- privitor la structuri:

Avem:

```
struct{
        char x;
        int y;
}
```

Ca și mai sus, nu putem face o afirmație privind modul în care sunt așezate în memorie variabilele din interiorul structurii. În plus, ca urmare a alinierii diferite după caz în memorie, nu putem face, în general presupunerea că:

sizeof(struct s) = sizeof(char) + sizeof(int)

OBSERVAȚIE:

Anumite procesoare nu acceptă descărcarea de regiștrii decât la adrese bine aliniate. (Pe Intel merge, dar mai lent).

Avem, următoarea situație, în care pe anumite procesoare am primi semnalul SIGBUS, semnificând o eroare de magistrală:

```
char buf[100];
char *p;
int x;
x=*(int*)(buf+1);        /* buf se incrementează cu 1 ca adresă, iar apoi se descarcă 4 octeți de la
                            adresa respectivă (care nu mai este multiplu de 4); moment în care se
                            primește SIGBUS */
```

De aceea, atenție in general la casting.

SIGFPE

- semnalează floating point exception;

OBSERVAȚIE:

Reamintim, că implicit comportamentul semnalelor este terminarea procesului.

APELURI SISTEM PENTRU LUCRUL CU SEMNALE

Blocarea semnalelor

Reamintim că procesele se comportă la primirea unui semnal, ca în cazul întreruperilor.
Dacă primim un semnal blocat, acesta devine **pending** (agățat), până în momentul deblocării, când este livrat procesului. În cazul în care mai multe semnale se află in stare pending, în momentul deblocării, se livrează doar ultimul.

OBSERVAȚIE:
Cele trei semnale nemodificabile (SIGSUSP, SIGCONT, SIGKILL) nu pot fi blocate.

În header-ul „**signal.h**" este definit tipul de date **sigset_t**. Acesta reprezintă o mulțime de semnale. Ni-l putem imagina ca pe un vector de n biți, unde biții 0 semnifică neapartenența semnalului la mulțime.

Avem următoarele primitive (nu sunt apeluri sistem):

- **int sigemptyset (sigset_t * p)**

- inițializează **p** cu mulțimea vidă de semnale;
- cod de retur: întotdeauna 0;

- **int sigfillset (sigset_t * p)**
-
  - inițializează **p** cu mulțimea totală (toți biții sunt 1);

- **int sigaddset (sigset_t * p, int sig)**

- **sig** este identificatorul unui semnal;
- adaugă semnalul **sig** la mulțimea **p**;
- dacă bitul corespunzător semnalului respectiv este deja 1, nu se modifică nimic;

- **int sigdelset (sigset_t * p, int sig)**

-
  - **sig** este identificatorul unui semnal;
  - elimină semnalul **sig** din mulțimea **p**;
  - dacă bitul corespunzător semnalului respectiv este deja 0, nu se modifică nimic;

- **int sigismember (sigset_t * p, int sig)**

  - **sig** este identificatorul unui semnal;
  - returnează: 1, dacă semnalul **sig** aparține mulțimii **p** sau
    0, în caz contrar;

Pentru blocarea/deblocarea semnalelor utilizăm apelul de sistem:

int sigprocmask (int op, sigset_t * p_nou, sigset_t * p_vechi)

- **op** indică operația pe care o realizăm, astfel poate avea valorile:

- **SIG_SETMASK** : caz în care **p_nou** va conține doar lista de semnale blocate, fără a ține cont de ce s-a întâmplat anterior;
- **SIG_BLOCK** : caz în care la lista anterioară se adaugă semnalele blocate din **p_nou**;
- **SIG_UNBLOCK** : semnalele din **p_nou** vor fi deblocate;

- **p_nou** reprezintă mulțimea desemnată pentru a fi blocată/deblocată;
- **p_vechi** reprezintă mulțimea semnalelor deja blocate;
- ambii pointeri trebuie fie sa fie adrese valide, fie NULL;

se disting două cazuri:

- **p_nou** este NULL, caz în care vrem să aflăm lista semnalelor blocate și nu modificăm nimic;
- **p_vechi** este NULL, caz în care nu ne interesează decât să facem modificarea;

Instalarea handlerelor scrise de noi

- în acest caz se intră în usermode când se execută funcția noastră;
- funcția pentru tratarea semnalelor este de forma:

```
void hand (int sig);    /*trebuie sa aibă parametrul int care pasează indicatorul
                          semnalului care a dus la acest apel. */
```

- exista două moduri de instalare a unui handler scris de noi:

1. modul simplu, prin apelul:

   void (* signal (int sig, void (* hand (int))))(int)

   - **sig** indică semnalul;
   - al doilea parametru este funcția pentru tratarea semnalelor de mai sus;
   - codul de retur este un pointer către o funcție de tip **void**; (această funcție era handlerul de semnal de până acum)

OBSERVAȚIE:

La prima întâlnire a semnalului se folosește handlerul nostru, însă apoi se restaurează handlerul implicit. Deci programul moare. De aceea, avem:

```
void hand (int sig){
        signal (sig, hand);      //comportament permanent
        .........
}
```

– **hand** poate avea valoarea unor pointeri valizi sau a unor constante de tip pointer, precum: - **SIG_DFL** - se restaurează comportamentul implicit
- **SIG_IGN** - un simplu **return,** nu face nimic;

2. modul complicat, prin apelul:

int sigaction (int sig, struct sigaction * p_nou, struct sigaction * p_vechi)

– **sig** indică semnalul;
– **p_nou** indică noul comportament, iar
– **p_vechi** este utilizat pentru a recupera situația anterioară;
– **p_nou** și **p_vechi** pot fi și NULL, ca și la **sigprocmask;**
– tipul **struct sigaction:**

```
struct sigaction{
        void ( * sa_handler) (int);     //pointer către fcț void
        sigset_t * sa_mask;             /* lista de semnale ce vreau să
                                           fie blocate pe timpul execuției
                                           handlerului.*/
        int sa_flags;                   // 0 sau SA_RESETHAND
}
```

- prin aceest apel se instaurează un comportament definitiv;
- **SA_RESETHAND** face ca acest comportament să fie valabil doar pentru prima întâlnire a semnalului;

Discutăm următoarea problemă: ne dorim să sărim dintr-un punct al programului într-o altă funcție.

Headerul **setjmp.h** conține tipul de date **sigjmp_buf**, care memorează un context de proces într-un anumit punct.
Pentru a-l inițializa, folosim apelul:

int sigsetjmp (sigjmp_buf buf, int opt)

- adaugă în **buf** contextul procesului în care s-a făcut apelul.
- **opt –** pentru a reține și masca de semnale blocate;

În handler, folosim apelul:

int siglongjmp (sigjmp_buf buf , int val)

- se reîntoarce în punctul în care s-a făcut **sigsetjmp**
- apelul nu poate fi făcut decât într-o funcție care e apelată direct sau indirect (descendentă) de **sigsetjmp;**
- val pasează o valoare, care trebuie să fie diferită de **0**, pentru a semnala că venim din **siglongjmp;** putem pasa valori care să ne indice în urma cărui semnal s-a făcut saltul;

ex.:
```
int x;
.......
x=sigsetjmp(buf,0);
//instrucțiuni
switch (x) {
        case 0: //normal
        case 1: //un anumit semnal, ex. SIGSEF
        .....
}
```

PROBLEME CLASICE DE CONCURENTĂ
(cazul general, nu doar UNIX)

Problema Producator – Consumator
(d.p.d.v. al proceselor)

Presupunem că procesul producător pune în memoria partajată, iar cel consumator ia /citește de acolo. Deducem că avem nevoie de următoarele date partajate:

- un buffer, care trebuie să fie circular;
- adresa la care va scrie procesul producător;
- adresa de la care va citi procesul consumator.

Situațiile în care procesele așteaptă unul după altul, după cum bufferul este plin sau gol, trebuiesc semaforizate, pentru a nu opri procesele.

Exemplu de implementare al acestui procedeu cu semafoare (tip Dijkstra):

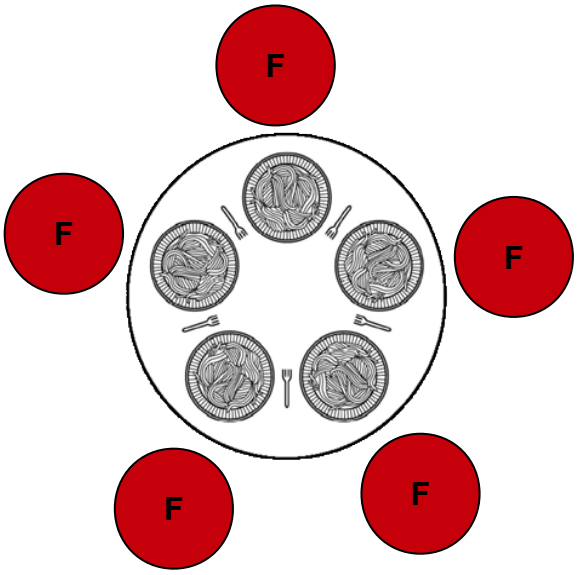
```
#define N 100
typedef int semaphore;  //acesta este de fapt un obiect sistem nu are tipul int
semaphore mutex = 1;  //împiedică un proces să intre în zona critică, când alt proces este acolo
semaphore empty = N;  //memorează numarul de locuri libere din buffer
semaphore full = 0;       //memorează numărul de locuri ocupate din buffer

void producer (){
        int item;
        while (TRUE){
                item=produce_item;      //produce următorul element;
                down (&empty);          /*decrementează numărul de locuri ocupate;
                                        dacă empty=0, rezultă că bufferul e plin, deci
                                        aștept.*/
                down (&mutex);          //intru în zona critică
                insert_item (item);
                up (&mutex);            //ies din zona critică
                up (&full);             //incrementează numărul de locuri ocupate
        }
}
        void consumer(){
        int item;
        while (TRUE){
                down (&full);           /*decrementează numărul de locuri ocupate;
                                        dacă bufferul e plin, atunci aătept */
                down (&mutex);
                item=remove_item();
                up (&mutex);
                up (&empty);            //incrementează numarul de locuri libere
                consume_item(item);
        }
```

}
OBSERVAȚIE: În UNIX, putem utiliza message queues

Punerea unor monitoare pe obiecte

*Java (final de semestru)

Problema celor 5 filosofi (n)



La o masă se află 5 filosofi, care doresc să mănânce. Pe masă se alfă farfurii și tacâmuri ca în imagine. Fiecare filosof are nevoie (evident) de două tacâmuri pentru a mânca.

Filosofii reprezintă procese, iar tacâmurile resurse utilizate de procese.

Se dorește evitarea situației de **deadlock.**