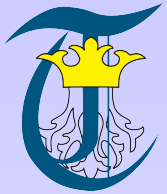

Universitatea *Transilvania* Brașov
Facultatea de Inginerie Electrică și Știința Calculatoarelor
Departamentul de Electronică și Calculatoare

Sisteme de operare

– curs 4 –

2013.03.20

ș.l. dr. ing. Kertész Csaba-Zoltán

Soluția lui Peterson

- prima soluție software pentru excluderea mutuală fără alternare strictă a fost dat de Dekker în 1965
  - introducerea unui flag prin care se semnalizează intenția de intrare în secțiune critică
- în 1981 Peterson a introdus un algoritm mai simplu pentru excludere mutuală cu busy-waiting

Algoritmul Peterson

```
#define FALSE 0
#define TRUE  1
#define N     2 //numărul proceselor

int turn;
int interested[N]; //inițial toate valorile sunt 0

void enter_region( int process ) //process = procesul care intră
{
        int other; //numărul celuilalt proces
        other = 1 – process; //celălalt proces
        interested[process] = TRUE; //procesul nostru e interesat
        turn = process; //setez flag
        while (turn == process && interested[other] == TRUE);
}

void leave_region( int process ) //process = procesul care iese
{
        interested[process] = FALSE;
}
```

Algoritmul Peterson

- înainte de a intra în secțiunea critică fiecare proces apelează `enter_region` cu numărul său
- după părăsirea secțiunii critice vor apela `leave_region` pentru a permite celuilalt proces intrarea în secțiune critică

Algoritmul Peterson

Proces 0	Proces 1

`enter_region(0)`
`  other = 1;`
`  interested[0] = TRUE;`
`  turn = 0;`
	`enter_region(1)`
	`  other = 0;`
	`  interested[1] = TRUE;`
	`  turn = 1;`

`secțiunea critică`	`buclă`
`leave_region(0)` `  interested[0] = FALSE;`	`buclă`
	`secțiune critică`

Instrucțiunea TSL
Test & Set Lock

- este o instrucțiune ce necesită sprijinul hardware (să se execute într-o singură instrucțiune)
- există procesoare care implementează în setul de instrucțiuni de tip TSL
- citește conținutul unui cuvânt din memorie într-un registru și apoi stochează acolo o valoare diferită
- procesorul garantează că instrucțiunea este indivizibilă

Instrucțiunea TSL

- echivalent cu

```
if (lock == 0)
    lock = 1;
```

- procesorul asigură și blocarea magistralelor pentru a preveni alt procesor să întrerupă operația
- algoritmul:

```
enter_region:
    tsl register, lock
    cmp register, #0
    jnz enter_region
    ret
```

```
leave_region:
    mov lock, #0
    ret
```

2.2.4. Sleep & wakeup

- soluția Peterson și TSL au 2 neajunsuri importante:
  - busy-waiting
  - inversarea priorităților:
    - dacă două procese având priorități diferite (un proces de prioritate mare H și un proces de prioritate mică L) sunt în stare de excludere mutuală, iar L intră în secțiunea critică, H poate fi planificat în timpul acestei secțiuni și să intre în busy-waiting din care nu mai iese niciodată, pentru că L nu poate fi planificat din cauza priorităților

Sleep & wakeup

- pentru a previne aceste neajunsuri se definesc două primitive IPC:
  - **sleep**

    apel sistem care determină blocarea procesului
  - **wakeup**

    semnal intern care deblochează procesul
- între blocare și deblocare pot fi planificate alte procese, astfel nu se folosește inutil procesorul, iar un proces de prioritate ridicată nu va mai bloca tot sistemul

Problema producător / consumator

- două procese partajează un buffer de mărime fixă
  - **producător**: plasează informații în buffer
  - **consumator**: preia informații din buffer
- problema apare când producătorul dorește să plaseze în buffer un articol, dar bufferul este deja plin
- soluția: producătorul execută sleep, iar când consumatorul preia un articol, va genera un semnal de wakeup
- analog și consumatorul

Problema producător / consumator

- abordarea sleep&wakeup este o soluție simplă dar poate conduce la apariția condițiilor de concurență (de ex. spooler)
- pentru a cunoaște starea bufferului e nevoie de o variabilă (**count**) <N
- producătorul testează count, dacă count==N, execută sleep, altfel pune articolul în buffer și count++
- consumatorul testează count, dacă count==0, execută sleep, altfel count--

Implementare producător / consumator

```
#define N 100

int count = 0;

void producer(void)
{
   int item;
   while (TRUE) {
      produce_item(&item);
      if (count==N)
         sleep();
      enter_item(item);
      count++;
      if (count==1)
         wakeup(consumer);
   }
}
```

```
void consumer(void)
{
   int item;
   while (TRUE) {
      if (count==0)
         sleep();
      remove_item(&item);
      count--;
      if (count==N-1)
         wakeup(producer);
      consume_item(item);
   }
}
```

Condiția de concurență

- apare la accesarea variabilei count:
  - bufferul e gol și consumatorul citește count
  - planificatorul întrerupe consumatorul și planifică producătorul
  - producătorul plasează un articol în buffer și văzând că count a devenit 1, emite un wakeup (care este pierdut pentru că consumatorul nu a executat sleep)
  - după ce consumatorul va fi planificat din nou, testează count, care e 0 și execută sleep (din care nu va mai fi trezit)

Wakeup întârziat

- poate rezolva problema, pentru că nu permite semnalul wakeup să fie pierdut
- dacă se emite un semnal wakeup către un proces care este „treaz” atunci se setează un flag special
- când procesul căruia a fost setat flagul execută sleep, aceasta nu se mai execută, ci doar flagul va fi resetat
- nu rezolvă problema în totalitate pentru că wakeup-urile multiple se vor pierde

2.2.5. Semafoare

- în 1965 Dijkstra a propus folosirea unei variabile pentru contorizarea wakeup-urilor salvate pentru viitoarea utilizare:

semafor

- semafor = 0: nici un wakeup nu a fost salvat
- semafor = n > 0: n wakeup-uri au fost salvate
- se folosesc două proceduri: **down** și **up** (care sunt practic generalizări ale apelurilor **sleep** și **wakeup**)

down

- verifică valoarea semaforului
- dacă semafor > 0, decrementează semaforul și continuă execuția procesului
- dacă semafor == 0, execută sleep
- verificarea valorii, modificarea și executarea sleep (dacă e cazul) formează o **operație atomică** (o operație indivizibilă) => pe durata executării acestei operații nici un alt proces nu are acces la semafor
- atomicitate este esențială pentru evitarea condițiilor de concurență

up

- incrementează valoarea semaforului
- dacă unul sau mai multe procese executau sleep determinat de semafor (incapabil să execute **down**), unul din ele va fi ales de sistem și va executa down
- după **up** semaforul poate să rămână 0, dar vor fi mai puține procese blocate de acel semafor
- **up** este de asemenea o operație atomică
- **up** nu poate bloca procesul respectiv

Problema producător / consumator rezolvat cu semafoare

- semafoarele vor rezolva problema pierderii semnalelor **wakeup**
- operații **up** și **down** vor fi implementate ca apel sistem
- pentru rezolvarea problemei e nevoie de 3 semafoare:
  - **full**: contorizarea pozițiilor ocupate
  - **empty**: contorizarea pozițiilor libere
  - **mutex**: excludere mutuală pentru accesul la buffer

Implementare prod. / cons. cu semafoare

```
#define N 100

typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
   int item;
   while (TRUE) {
      produce_item(&item);
      down(&empty);
      down(&mutex);
      enter_item(item);
      up(&mutex);
      up(&full);
   }
}
```

```
void consumer(void)
{
   int item;
   while (TRUE) {
      down(&full);
      down(&mutex);
      remove_item(&item);
      up(&mutex);
      up(&empty);
      consume_item(item);
   }
}
```

Moduri de utilizare a semafoarelor

- full și empty sunt folosite pentru a asigura respectarea limitelor bufferului
- mutex: pentru excludere mutuală
  - dacă semaforul este inițializat cu 1 și este utilizat de unul sau mai multe procese pentru a se asigura accesul numai a unei dintre ele la o secțiune critică se numește **semafor binar**
  - dacă înainte de fiecare secțiune critică se va executa un **down**, iar după secțiune un **up** atunci excluderea mutuală este garantată

2.2.6. Contoare de evenimente

- soluția problemei prod./cons. cu semafoare s-a bazat pe excluderea mutuală pentru evitarea condiției de concurență
- există însă și o soluție care nu necesită excludere mutuală prin utilizarea unui tip special de variabilă:

contor de evenimente

- se definesc 3 operații:
  - read(E): returnează valoarea curentă a lui E
  - advance(E): incrementează atomic E
  - await(E, v): așteaptă până când E >= v

Implementare prod. / cons. cu contoare de evenimente

```
#define N 100

typedef int ev_counter;

ev_counter in = 0;
ev_counter out = 0;

void producer(void)
{
   int item, seq=0;
   while (TRUE) {
      produce_item(&item);
      seq++;
      await(out, seq-N);
      enter_item(item);
      advance(&in);
   }
}
```

```
void consumer(void)
{
   int item, seq=0;
   while (TRUE) {
      seq++;
      await(in, seq);
      remove_item(&item);
      advance(&out);
      consume_item(item);
   }
}
```

2.2.7. Monitoare

- dacă la problema prod./cons. cu semafoare, operațiile down-down respectiv up-up sunt inversate poate apare situația blocării proceselor
- dacă mutex va fi decrementat înainte de empty și bufferul a fost plin, producătorul blochează mutexul, după care consumatorul face down și el la mutex, astfel blocându-se amândouă procese la mutex:

dead-lock

Monitoare

- pentru a ușura scrierea corectă a programelor Hoare ('74) și Hansen ('75) au propus o primitivă de nivel înalt numit **monitor**
- un monitor este o colecție de proceduri, variabile și structuri de date grupate într-un mod special
- procesele pot apela procedurile monitoarelor, dar nu pot accesa direct structurile de date interne ale monitoarelor

Exemplu de monitor

```
monitor example
   integer i;
   condition c;
   procedure producer(x);
   ...
   end;
   procedure consumer(x);
   ...
   end;
end monitor
```

Proprietăți ale monitoarelor

- numai un proces poate fi activ într-un monitor la un moment dat
- monitoarele sunt elemente ale unui limbaj de programare
- dacă un proces apelează o procedură a unui monitor, primele instrucțiuni ale procedurii respective vor verifica dacă există un alt proces activ în cadrul monitorului, dacă da procesul apelant va fi suspendat până când celălalt proces părăsește monitorul
- compilatorul implementează excluderea mutuală cu semafoare binare

Variabile de condiție

- excluderea mutuală (realizată de compilator) nu este suficient, este nevoie de o modalitate de a bloca un proces când nu poate realiza o acțiune (de ex. teste pentru buffer full sau empty)
- soluția este folosirea variabilelor de condiție împreună cu două operații asociate:
  - wait
  - signal

Variabile de condiție

- când o procedură monitor descoperă că nu poate continua va executa un **wait** asupra unei variabile de condiție, care va bloca procesul apelant și va permite altor procese să intre în monitor
- celălalt proces poate trezi procesul blocat prin execuția unui **signal** asupra variabila de condiție
- signal trebuie să fie ultima instrucțiune într-un monitor

Dezavantaje ale monitoarelor

- variabilele de condiție nu sunt contoare, deci semnalele pot fi pierdute
- sunt puține limbaje care implementează monitoare (de ex. Concurent Euclid)
  - în alte limbaje vor trebui implementate rutine în asamblare
- în cazul în care există mai multe CPU și nu există memorie partajată (sistem distribuite)
  semafoarele și monitoarele nu pot fi folosite

Implementare prod. / cons. cu monitoare

```
monitor ProducerConsumer
   condition full, empty;
   integer count;
   procedure enter;
   begin
      if count=N then wait(full)
      enter_item;
      count := count+1;
      if count=1 then signal(empty)
   end;
   procedure remove;
   begin
      if count=0 then wait(empty)
      remove_item;
      count := count-1;
      if count=N-1 then signal(full)
   end
   count := 0;
end monitor;
```

```
procedure producer;
begin
   while true do
   begin
      produce_item;
      ProducerConsumer.enter;
   end
end;

procedure consumer;
begin
   while true do
   begin
      ProducerConsumer.remove;
      consume_item;
   end
end;
```

2.2.8. Message passing

- dacă procesele se află pe două mașini diferite, sincronizarea între ele se poate face prin transmisia și recepția unor mesaje
- se utilizează două primitive (apeluri sistem)
  - `send (destination, &message);`
    - neblocant
  - `receive (source, &message);`
    - blochează când nu există nici un mesaj

Mecanismele message-passing

- mesajele transmise prin rețea pot fi pierdute, de aceea receptorul trimite un mesaj de confirmare la transmițător, în lipsa confirmării transmițătorul retrimite mesajul
- dacă se pierde confirmarea, transmițătorul va transmite din nou, iar receptorul va trebui să distingă între mesajele diferite și copiile mesajelor
  - trebuie să existe o metodă de marcare a mesajelor
- trebuie să existe posibilitate identificării transmițătorului și receptorului
- trebuie rezolvat problema autentificării transmițătorului

Problema prod. / cons. cu message-passing

```
#define N 100
#define MSIZE 4

typedef int message[MSIZE];

void producer(void)
{
   int item;
   message m;


   while (TRUE) {
      produce_item(&item);
      receive(consumer, &m); //empty
      build_message(&m, item);
      send(consumer, &m);
   }
}
```

```
void consumer(void)
{
   int item, i;
   message m;
   for (i=0; i<N; i++)
      send(consumer, &m); //empty
   while (TRUE) {

      receive(producer, &m);
      extract_item(&m, &item);
      send(producer, &m);
      consume_item(item);
   }
}
```

Problema prod. / cons. cu message-passing

- mesajele trimise dar nerecepționate sunt bufferate de SO
- avem în total N mesaje
- dacă producătorul a create elementul, va prelua un mesaj empty și va trimite un mesaj full
- dacă nu este nici un mesaj empty recepția va bloca
- numărul de mesaje este constant => putem folosi buffer finit pentru stocarea mesajelor

Aspecte message-passing

- există 2 posibilități de implementare:
  - utilizând mailbox-uri
    - fiecare proces va avea un mailbox cu capacitatea de N mesaje
  - rendezvous
    - nu există nici un fel de bufferare
- semafoarele, monitoarele și message-passing sunt echivalente: putem utiliza una din ele pentru a implementa oricare alta