Curs 6

Sincronizari

SO = deterministe (acelasi output pt acelasi input)
= nedeterministe (din cauza evenimentelor externe)

Definitii

Procese CONCURENTE = procese ce se executa simultan in sistem
Procese = asincrone <=> nu sunt garantii despre starea unui proces la un timp t
Procese independente = procese care nu partajeaza resurse
Procese dependente = partajeaza resurse pt a-si indeplini obiectivele (exemplu imprimanta)
Sectiune/regiune critica = portiune de cod care acceaseaza o resursa partajata

Race condition = situatie in care executia intresetusa a mai multor procese care acceseaza o resursa partajata induce rezultate nedeterministe

Excludere mutuala = situatie in care cel mult un proces are acces la un moment dat la o resursa partajata

Deadlock = sitautie in care niciun proces nu poate continua din cauza resurselor necesare care sunt detinute de alt proces

Sincronizare = cerinta ca un proces sa fi atins o anumita etapa in calcul sau inainte ca al proces sa poata continua

Starvation = resuresele necesare unui proces nu ii sunt niciodata puse la dispozitie (exemplu un proces de prioritate mica e impiedicat sa acceseze o resursa care e constant obtinuta de un proces cu prioritate mare)

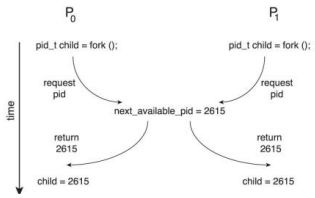
OBS: daca proecsele concurente evita race-condition uri, deadlock uri si starvation => resursele partajate sunt folosite corect. In acest scop avem notiunile: **SECTIUNI CRITICE, EXCLUDERE MUTUALA, SINCRONIZARE**.

Cerintele necesare unei solutii corecte de partajare a resurselor de procese corecte: FARA PRESPUNERI DE VITEZA A EXECUTIEI PROCESELOR CONCURENTE, EXCLUDERE MUTUALA, ASTEPTARE LIMITATA(un proces care solitica accesul in C.S. trebuie sa-l obtina candva), PROGRES(un proces care ruleaza inafara CS nu trebuie sa blocheze unul care intra in CS)

Exemplu pentru RACE CONDITION

Processes P_0 and P_1 are creating child processes using the `fork()` system call

Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



Critical Section Problem

Considerand N procese (p0, ..., pn) si presupunem ca fiecare intra in sectiunea sa critica, atunci vor putea aparea modificari nedorite care s-ar putea sa produca rezultate nedeterministe. Astfel, inainte de a intra in seciunea critica, fiecare proces ar trebui mai intai sa obtina permisiunea. Structura generala pentru CS:

```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

Pentru a rezolva CS Problem folosim: excludere mutuala (cel mult un proces acceseaza CS), progres (fiecare proces care vrea sa intre in CS la un moment dat trebuie sa intre garantat) si limita de asteptare (un numar limitat de ori pentru restul proceselor care intra in CS fata de cel care a cerut acces in CS).

```
int turn = 0;
P0 ()
{
        while(1)
        {
                while(turn != 0)
                        ;
                sectiune_critica();
                turn = 1;
                sectiune_necritica();
        }
}
```

```
P1 ()
{
        while(1)
        {
                while(turn != 1)
                        ;
                sectiune_critica();
                turn = 0;
                sectiune_necritica();
        }
}
```

Solutia anterioara (ALTERNANTA STRICTA) nu respecta cerinta de PROGRES deoarece daca unul dintre procese petrece mult timp inafara CS, celalalt proces e in mod nejustificat impiedicat sa intre in CS.

Algoritmul lui Peterson
satisface excluzie mutuala, progres, si bounded waiting

```
const int N = 2;
bool flag[N];
int turn;
void func(int i)
{
        j = 1 – i;
        flag[i] = true;
        turn = j;
        while(flag[j] && (turn == j))
                ;
        sectiune_critica()
        flag[i] = false;
        sectiune_necritica();
}
```

Pentru a ne asigura ca Algoritmul lui Peterson functioneaza si pe arhitecturile moderne de calcul vom folosi **Memory Barrier** (care reprezinta o instructiune ce forteaza orice schimbare in memorie sa fie vizibila pentru toate procesoarele).

Test_and_set

```
Boolean test_and_set(boolean *target){
        Boolean rv = *target;
        *target = true; #seteaza true in parametrul functiei
        Return rv;     #returneaza val originala a parametrului
}
```

Spinlocks cu TAS

Ofera doua operatii: acquire
(la intrarea in CS) si release (la iesirea din CS).
Cand un proces detine lock-ul celelalt
e sunt in busy waiting.

```
int lock = 0;
void acquire(int *lock)
{
        while(tas(lock))
                ;
}
void release(int *lock)
{
        *lock = 0;
}
```

Compare and swap

```
int compare_and_swap(int *value, int expected, int new_value){
        int temp = *value;
        if (*value == expected)
                *value = new_value;
        return temp;
}
```

Solutii cu TAS sau CAS:

```
while (true){
        while (CONDITIE)  # compare_and_swap(&lock, 0, 1) != 0 SAU
                                                        #(test_and_set(&lock)
                ; /* do nothing */

        /* critical section */

        lock = 0;

        /* remainder section */
}
```

Variabila atomica = da update-uri atomice (neintrerupte).
Exemplu pt functia increment (incrementeaza o variabila fara intreruperi):

```
void increment(atomic_int *v) {
        int temp;
        do {
                temp = *v; }
         while (temp != (compare_and_swap(v,temp,temp+1));
        }
```

TAS/CAS functioneaza pe procesoare CISC, dar nu pe RISC. Pe procesoare de tip RISC se vor folosi operatii speciale precum Load Linked (LL) si Store Conditional (SC).

Load linked functioneaza astfel: incarca o variabila din memorie intr-un registru CPU si apoi verifica activ daca variabila din memorie e modificata de alte procesoare.

Store Conditional verifica daca au existat modificari ale variabilei de memorie de la ultimul LL. (daca da => val registrului = 0, altfel = 1, iar variabila din memorie e modifcata)

Exemplu de incrementare atomica (asemantor cu CAS si TAS)

```
1   increment:
2       ll      $2, count           ; incarca valoarea counter-ului
3       addu    $3, $2, 1           ; incrementeaza valoare counter
4       sc      $3, count           ; incearca sa stocheze noua valoare
5       beq     $3, zero, increment         ; zero inseamna esec
6       j       $31                 ; return din rutina
```

Proprietati ACID

Atomicitate = modificarile sunt executate ca si cand operatia ar fi atomica (ori toate modificarile sunt executate, ori niciuna)

Consistenta = datele sunt intr-o stare consistenta cand tranzactia incepe si cand se termina (counterul e incrementat corect, chiar daca se executa mai multe tranzactii simultan)

Izolare = starea intermediara a tranzactiei e invizibila altor tranzactii, astfel tranzactiile concurente par a fi serializate

Durabilitate = dupa incheierea cu succes a tranzactiei datele sunt salvate in memoria principala RAM si modificarea e finala

Mutex Locks

Software tool care ajuta la CS problem, prin functii precum acquire() si realease(), functii care sunt ATOMICE.

```
while (true) {
        acquire lock

            critical section

        release lock

remainder section
}
```

OBS: busy-waiting-ul se poate inlocui cu ajutorul unei primitive sleep

OBS: cand procesul aflat in sectiune critica paraseste sectiunea, apeleaza primitiva wakeup care trezeste toate procesele blocate in sleep

SEMAFOARE

Structura de date care implica un contor ce numara wake-upurile si o coada de procese blocate in sleep.

Operatia DOWN => decrementeaza atomic contorul daca e >0 si permite procesului sa continue, iar daca avem contorul = 0 blocheaza procesul si il pune in coada

Operatia UP => verifica daca exista procese blocate in coada, daca da alege unul si il deblocheaza, altfel incrementeaza atomic contorul

OBS: un semafor cu un contor initial 1 = mutex (semafor binar)

Up/down generalizeaza sleep/wakeup dar intr-un mod ATOMIC

Up nu blocheaza niciodata procese

Doua procese nu pot face UP/DOWN pe acelasi semafor

Exemplu de semafor in cod:

typedef struct {
    int value;
    struct process *list;} semaphore;

```
down(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

```
up(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Monitors

Un monitor repreiznta un element abstract care ofera un mecanism eficient pentru sincronizarea proceselor. Un singur proces e activ in monitor la un anumit timp.

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure P2 (…) { …. }

  procedure Pn (…) {……}

  initialization code (…) { … }
}
```

Monitorul contine proceduri pentru a gestiona sincronizarea proceselor. Acestea sunt instrumentate de compialtor sa execute o secventa speciala de apelare. Cand un proces apeleaza o procedura din monitor:

Se verifica daca exista alt proces activ in interiorul monitorului
Daca da => procesul e suspendat pana cand se termina cel din monitor
Daca nu => procesul apelant poate intra in monitor

OBS: implementarea se face in mod uzual folosind un semafor
Compilatorul genereaza automat cod de excludere mutuala

Daca un proces din monitor trebuie blocat apelam la **VARIABILE CONDITIE**. Acestea suporta doua operatii:
Wait -> blocheaza procesul apelant si elibereaza monitorul pt ca alte procese blocate la intrarea in monitor sa poate accesa monitorul
Signal -> trezeste procesul care a apelat anterior wait pe aceasta variabila conditie

Problema signal = daca trezeste un proces ce apelase anterior wait vor exista 2 procese simult in monitor, iar pt a se evita aceasta se foloseste solutia HANSEN (signal nu se poate executa decat ca ultima operatie a unei proceduri de monitor).
OBS: wait se executa inainte de signal!

Monitoarele sunt un concept de nivel de limbaj de programare, spre deosebire de semafoare care pot fi implementate si ca apeluri de biblioteca (cu suport din partea sistemului de operare, evident).

Atat semafoarele cat si monitoarele functioneaza doar pe sisteme cu memorie partajata (inclusiv multiprocesoare), NU pe sisteme distribuite.

Exemplu de implementare semafoare cu monitoare in Java:

```
public class semaphore
{       unsigned int counter;
        public synchronized void down()
        {
                if(counter > 0)
                        { counter--, return; }
                wait();
        }
        public synchronized void up()
        {       counter++, notify();
        }
}
```

Liveness = set de reguli pe care un sistem trebuie sa le indeplineasca pentru a asigura progresul

Exemplu de DEADLOCK:

P_0	P_1
down (S) ;	down (Q) ;
down (Q) ;	down (S) ;
. . .	. . .
up (S) ;	up (Q) ;
up (Q) ;	up (S) ;

Conditii necesare pt DEADLOCK:

1. Excludere mutuala
2. Hold & wait (procesul detine o resursa si asteapta altele de la alte proc.)
3. No preemption (resursele unui proces nu pot fi confiscate de alt proces)
4. Asteptare circulara

Daca 1,2,3,4 sunt indeplinite => DEADLOCK

Alte forme de deadlock:
Starvation: un proces nu este eliminat din coada de asteptare a unui semafor

Priority Inversion: procesele cu prioritate scazuta blocheaza procese cu prioritate mare (incident rezolvat de priority-inheritance protocol)