Tutoriat 6 SO

Sincronizare

Motivatie?

Accesul simultan la date poate rezulta în inconsistenta acestora.

Producer - Consumer Problem

In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

Producer

```
while (true) {
       /* produce an item in next produced */

       while (counter == BUFFER_SIZE) ;
              /* do nothing */
       buffer[in] = next_produced;
       in = (in + 1) % BUFFER_SIZE;
       counter++;
}
```

Operating System Concepts – 9th Edition 5.5 Silberschatz, Galvin and Gagne ©2013

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```


Care e problema cu abordarea aceasta?

Dacă ambele procese modifica variabila counter simultan, poate apărea următorul scenariu:

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute	`register1 = counter`	{register1 = 5}
S1: producer execute	`register1 = register1 + 1`	{register1 = 6}
S2: consumer execute	`register2 = counter`	{register2 = 5}
S3: consumer execute	`register2 = register2 - 1`	{register2 = 4}
S4: producer execute	`counter = register1`	{counter = 6 }
S5: consumer execute	`counter = register2`	{counter = 4}

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

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

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

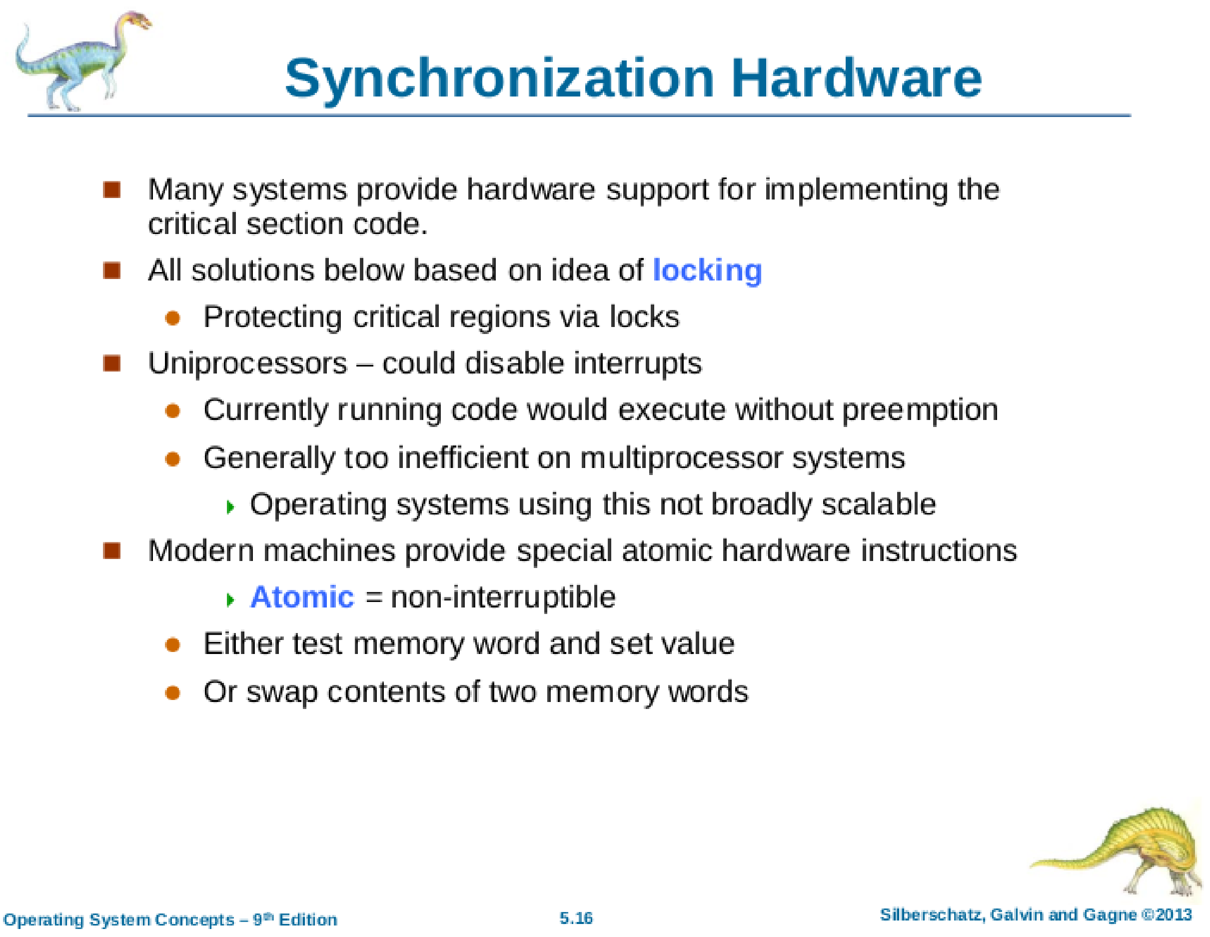
1. Excluziunea mutuala - dacă un proces Pi se executa în critical section, atunci niciun alt proces nu se poate executa în critical section.
2. Progresul - Dacă niciun proces nu se executa în critical section și există niște procese care vor sa intre in critical section atunci selecția procesului care sa intre nu poate fi amânată infinit. (dacă nu e nimeni în critical section dar sunt procese care vor sa intre atunci trebuie să intre cineva).
3. Timpul finit de așteptare - Orice proces din coada va aștepta un timp finit până să intre în critical section

Dacă nu avem Progres => Deadlock (Nimeni nu intra în zona critică și toate procesele se blochează la intrare)

Dacă nu avem Timp finit de așteptare => Putem intra pentru un set de procese în Starvation (adică ele nu vor apuca niciodată să se execute)

Preemptive => un proces se poate întrerupe oricând
Non-preemptive => un proces nu se poate scoate de pe procesor până nu termină

Instrucțiune atomica => instrucțiune hardware care nu poate fi întreruptă (sau sparta în instrucțiuni mai mici)



test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
  {
       boolean rv = *target;
       *target = TRUE;
       return rv:
  }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Se executa atomic, returneaza valoarea originala primita ca parametru si seteaza valoarea parametrului la TRUE.

Soluția pentru critical section problem cu test_and_set():

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock))
       ; /* do nothing */
          /* critical section */
    lock = false;
          /* remainder section */
 } while (true);
```

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Se executa atomic, returneaza valoare originala primita ca parametru prin value, seteaza apoi variabila value cu valoarea new_value dar doar dacă value == expected.

Soluția pentru critical section problem cu compare_and_swap:

- Shared integer “lock” initialized to 0;
- Solution:

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0)
       ; /* do nothing */
    /* critical section */
 lock = 0;
    /* remainder section */
} while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

- ```
  acquire() {
     while (!available)
        ; /* busy wait */
     available = false;;
  }
  ```
- ```
  release() {
     available = true;
  }
  ```
- ```
  do {
     acquire lock
        critical section
     release lock
        remainder section
  } while (true);
  ```
  ```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```


Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0 / P_1

```
P0                    P1
wait(S);              wait(Q);
wait(Q);              wait(S);
 ...                   ...
signal(S);            signal(Q);
signal(Q);            signal(S);
```

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Probleme clasice de sincronizare

1. Bounded-Buffer Problem
2. Readers and Writers Problem
3. Dining-Philosophers Problem

1. Bounded-Buffer Problem

Avem nevoie de 3 semafoare:
- mutex, inițializat cu 1
- full, inițializat cu 0 (semnifica spațiul ocupat din buffer)
- empty, inițializat cu n (semnifica spațiul liber din buffer)

Producer:

```
while(1) {
        produce()
        wait(empty)    // scade numărul de locuri libere pentru ca am
                       // produs ceva. Dacă bufferul e plin, empty = 0
                       // deci se va aștepta până când un consumer
                       // eliberează un spațiu
        wait(mutex)    // asigura ca niciun alt proces nu intra in crit. sect.
        append()       // critical section
        signal(mutex) // notifica ieșirea din critical section, a.i alte procese
                       // să poată intra
        signal(full)   // incrementeaza numărul de locuri ocupate
                      // după adaugarea noului produs (în append())

}
```

Consumer:

```
while(1) {
        wait(full)      // decrementeaza numărul de locuri ocupate
                        // din moment ce consumăm un produs
        wait(mutex)    // la fel ca la producer
        take()         // critical section
        signal(mutex)
        signal(empty) // incrementeaza numărul de locuri libere
}
```

2. Readers-Writers Problem

Avem nevoie de:

- un semafor w inițializat cu 1
- un semafor m inițializat cu 1
- un întreg read_count inițializat cu 0

Writer:

```
while(TRUE)
{
    wait(w);

   /* perform the write operation */

   signal(w);

}
```

Reader:

```
while(TRUE)
{
    //acquire lock
    wait(m);
    read_count++;
    if(read_count == 1)
        wait(w);

    //release lock
    signal(m);

    /* perform the reading operation */

    // acquire lock
    wait(m);
    read_count--;
    if(read_count == 0)
        signal(w);

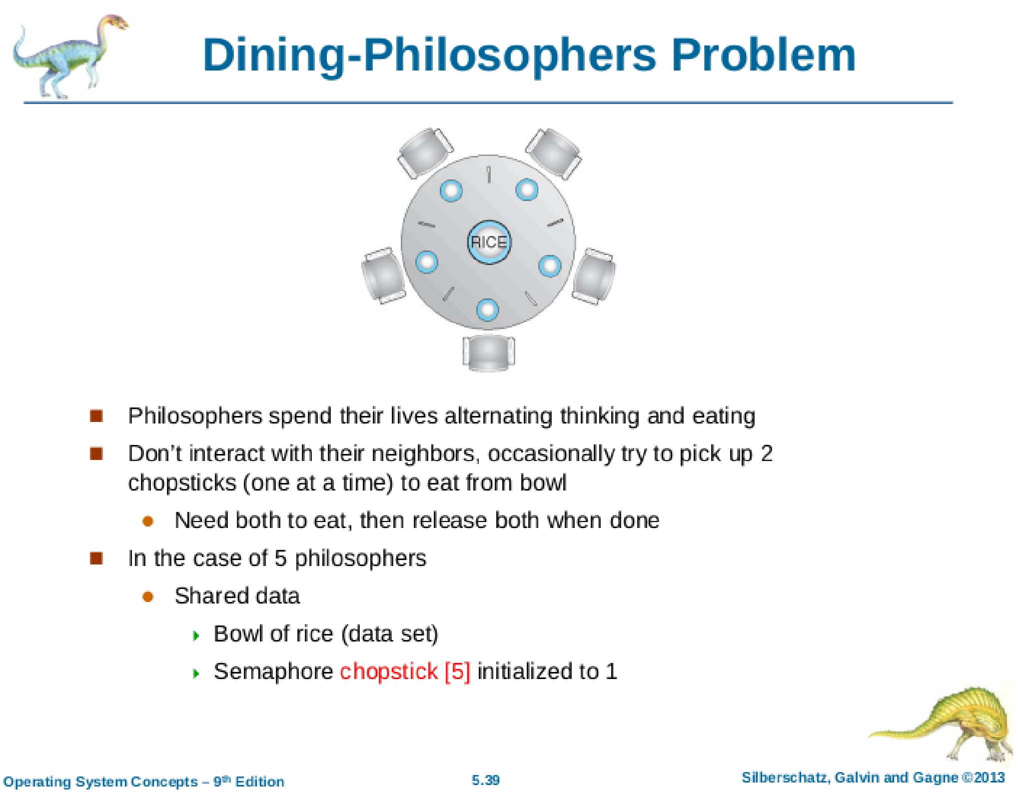
    // release lock
    signal(m);

}
```

- The semaphore w is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first reader enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the w semaphore because there are zero readers now and a writer can have the chance to access the resource.

Poate apărea starvation! (De exemplu, dacă în prezent se executa cititorii, un scriitor așteaptă în coada și tot vin alți cititori în capătul cozii, ei vor putea să intre în execuție iar scriitorul va putea intra doar cand toți cititorii au terminat).

3. Dining-Philosophers Problem



Pi:

```
do {
	wait(chopstick[i]) //stanga
	wait(chopstick[(i+1)%5 ]) //dreapta
	//eat
	signal(chopstick[i])
	signal(chopstick[(i+1)%5 ])
	//think
} while (true)
```

Care e problema?

Pe varianta de mai sus poate apărea deadlock dacă toți executa linia 1 în același timp.

Soluții:

- Au voie doar 4 filozofi la masa. De ce? Dacă fiecare ridica un chopstick, mai ramane unul disponibil, deci o persoana poate manca. După ce termina, avem 2 chopstick-uri disponibile, etc.
- Filozofii pot lua betisoarele doar dacă ambele sunt libere.
- Un filozof cu nr impar ridica mai intai stanga și după dreapta. Un filozof cu nr par ridica mai intai dreapta, apoi stanga

Scenariu:

Avem 100 de thread-uri și fiecare trebuie sa incrementeze de un număr de ori (spre exemplu 10000) o variabila globală și ne dorim ca la final, variabila sa aibă o valoare egala cu numărul de threaduri * numărul de pași per thread.

Fără sincronizare, comportamentul este unul nedefinit (la multiple rulări, valoarea finală variază). Exemplu:

```
cosmin@cosmin-Legion-
Count este 155734
cosmin@cosmin-Legion-
Count este 227213
cosmin@cosmin-Legion-
Count este 141346
cosmin@cosmin-Legion-
Count este 177803
cosmin@cosmin-Legion-
Count este 183325
cosmin@cosmin-Legion-
Count este 169869
cosmin@cosmin-Legion-
Count este 122294
cosmin@cosmin-Legion-
```

Soluția este să protejăm operația de incrementare prin intermediul unui mutex (sau semafor).

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>

#define NR_THREADS 100
#define MAX_INCREMENT 10000

int count = 0;

pthread_mutex_t mtx;

void *tfun(void *v)
{
```

```c
    for (int i = 0; i < MAX_INCREMENT; i++)
    {
        pthread_mutex_lock(&mtx);
        count++; //critical section
        pthread_mutex_unlock(&mtx);
    }
    return NULL;
}

int main()
{

    pthread_t thr[NR_THREADS];

    if (pthread_mutex_init(&mtx, NULL))
    {
        perror(NULL);
        return errno;
    }

    for (int i = 0; i < NR_THREADS; i++)
    {
        if (pthread_create(&thr[i], NULL, tfun, NULL))
        {
            perror(NULL);
            return errno;
        }
    }

    for (int i = 0; i < NR_THREADS; i++)
    {
        if (pthread_join(thr[i], NULL))
        {
            perror(NULL);
            return errno;
        }
    }

    printf("Count este %d\n", count);

    pthread_mutex_destroy(&mtx);

    return 0;
}
```

```
cosmin@cosmin-Legion
Count este 1000000
cosmin@cosmin-Legion
Count este 1000000
cosmin@cosmin-Legion
Count este 1000000
cosmin@cosmin-Legion
Count este 1000000
cosmin@cosmin-Legion
Count este 1000000
cosmin@cosmin-Legion
Count este 1000000
cosmin@cosmin-Legion
```

Similar, putem rezolva prin intermediul unui semafor:

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#include <semaphore.h>

#define NR_THREADS 100
#define MAX_INCREMENT 10000

int count = 0;

sem_t sem;

void *tfun(void *v)
{
    for (int i = 0; i < MAX_INCREMENT; i++)
    {
        if (sem_wait(&sem))
        {
            perror(NULL);
            return errno;
        }
        count++; //critical section
        if (sem_post(&sem))
        {
            perror(NULL);
            return errno;
        }
    }
```

```c
    return NULL;
}

int main()
{

    pthread_t thr[NR_THREADS];

    // OBS un mutex e un semafor cu s = 1

    int S = 1;
    if (sem_init(&sem, 0, S))
    {
        perror(NULL);
        return errno;
    }

    for (int i = 0; i < NR_THREADS; i++)
    {
        if (pthread_create(&thr[i], NULL, tfun, NULL))
        {
            perror(NULL);
            return errno;
        }
    }

    for (int i = 0; i < NR_THREADS; i++)
    {
        if (pthread_join(thr[i], NULL))
        {
            perror(NULL);
            return errno;
        }
    }

    printf("Count este %d\n", count);

    sem_destroy(&sem);

    return 0;
}
```

Useful links:

- The Producer-Consumer problem in Operating System.
- Readers Writer Problem in OS
- Dining Philosophers Problem