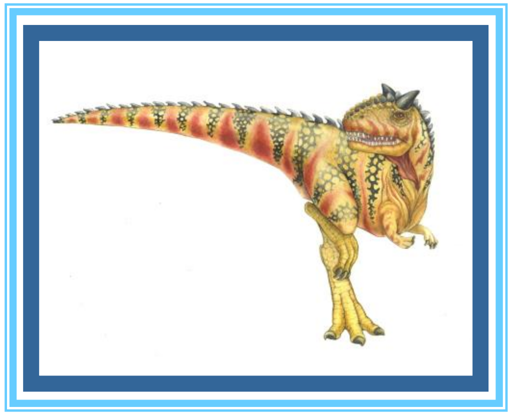
Chapter 6: Synchronization Tools



Outline

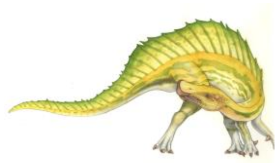
- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Objectives

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- We illustrated in chapter 4 the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.



Determinism in sistemele de operare

- SO sunt deopotriva deterministe si nedeterministe
- cand ruleaza un proces, garanteaza ca rularea va produce mereu aceleasi rezultate daca se furnizeaza aceleasi date de intrare
- natura impredictibila a evenimentelor externe (intreruperi sau cereri de a rula programe carora trebuie sa le raspunda) implica nedeterminism in privinta ordinii de executie a proceselor
- vom numi procese *concurente* procese care exista simultan in sistem, indiferent daca ruleaza time-shared pe un singur CPU (concurenta logica) sau in paralel pe mai multe CPU (concurenta fizica)
- data fiind nedeterminarea inerenta sistemului de operare, nu exista garantii despre starea unui proces la un moment dat
- din aceasta cauza, vom numi procesele *asincrone*

Definitii IPC

- procese *independente* – procese care nu partajeaza resurse
- procese *dependente* – partajeaza resurse pt a-si indeplini obiectivele
  - ex resursa partajata: imprimanta
  - prin intermediul primitivelor IPC, procesele dependente isi coordoneaza accesul la resursele patajate cf unui anumit protocol
- *sectiune/regiune critica* – portiune de cod care acceseaza o resursa partajata
- *race condition* – situatie in care executia intretesuta (interleaved) 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

Definitii IPC (cont’d)

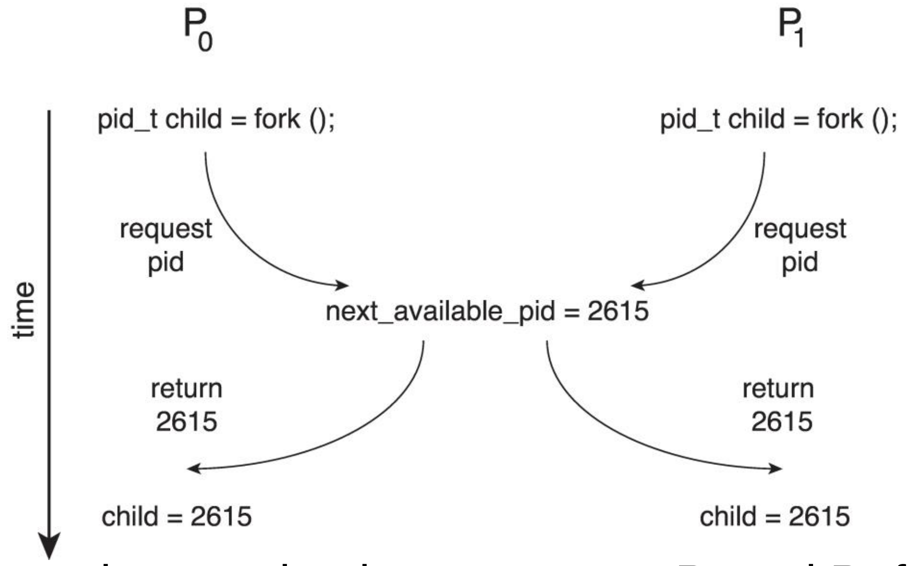
- *deadlock* – situatie in care nici un proces nu poate continua pt. ca resursele de care are nevoie sunt detinute de un alt proces
  - pp. implicit accesul mutual exclusiv la resursele partajate si o forma sau alta de asteptare circulara (ciclu intr-un graf de alocare a resurselor)
- *sincronizare* – cerinta ca un proces sa fi atins o anumita etapa in calculul sau inainte ca alt proces sa poata continua
- *starvation* – resursele necesare unui proces nu ii sunt niciodata puse la dispozitie
  - ex: un proces de prioritate mica P_l este impiedicat sa acceseze o resursa care e constant obtinuta de un proces de prioritate mare P_h
  - spunem ca P_l “moare de foame”

Corectitudinea partajarii resurselor

- resursele partajate sunt folosite corect daca procesele concurente evita race condition-uri, starvation si deadlock
- mecanismele utilizate in acest scop sunt: sectiunile critice, excluderea mutuala si sincronizarea
- cerinte necesare unei solutii corecte de partajare a resurselor de catre procese concurente
  - *nu e permisa nici o presupunere vis-à-vis de viteza relativa de executie a proceselor concurente*
  - ***excludere mutuala*** – cel mult un proces poate fi in sectiune critica la orice moment dat
  - ***asteptare limitata*** – daca un proces solicita accesul in sectiune critica trebuie sa i se garanteze ca-l va obtine candva
  - ***progres*** – un proces care ruleaza in afara sectiunii critice nu trebuie sa blocheze alt proces care vrea sa intre in sectiunea critica

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)



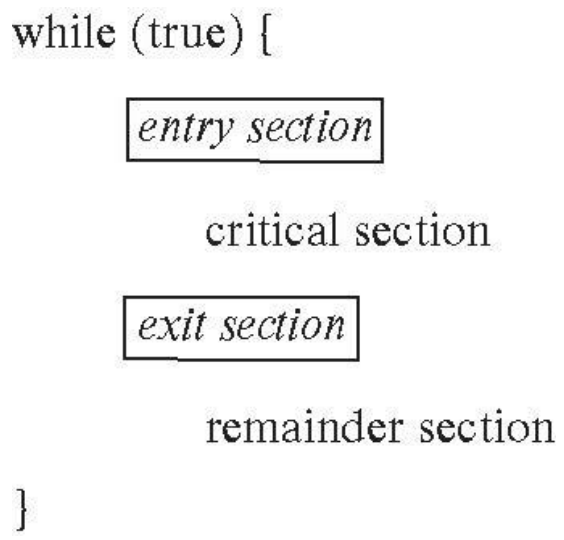
- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … 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**$_i$*



```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Critical-Section Problem (Cont.)

Requirements for 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 process 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

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?

Dezactivarea intreruperilor

- bazata pe instructiuni privilegiate (eg Intel cli & sti) inaccesibile din user-mode
- are ca efect cresterea latentei intreruperilor => afecteaza intregul sistem, inclusiv operatii de I/O fara legatura cu sectiunea critica
- daca se dezactiveaza intreruperea de ceas, e posibil ca unele procese sa nu ajunga sa fie planificate pt rulare pe procesor
- intr-un sistem multiprocesor metoda e ineficienta (nu se poate garanta ca alte procese concurente care vor sa intre in sectiune critica nu vor rula pe alte procesoare)
- tehnica utila in kernele uniprocesor pt sectiuni critice scurte

Software Solution 1

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **`int turn;`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- initially, the value of **`turn`** is set to *i*

Alternanta stricta

```
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();
        }
}
```

Observatii

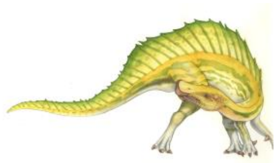
- procesele asteapta sa le vina randul sa intre in sectiunea critica ocupand procesorul (*busy waiting*)
  - pe un sistem uniprocesor time-shared, daca procesul aflat in sectiune critica pierde CPU, celalalt proces va fi planificat pt rulare si isi va consuma intreaga cuanta de timp alocata in busy waiting
  - pe sisteme multiprocesor, cele doua procese ruleaza in paralel pe CPU diferite si, cand unul, cand celalalt, consuma inutil cicluri procesor in busy waiting pt a intra in sectiune critica

  => din ratiuni de eficienta, sectiunile critice implementate in busy waiting trebuie sa fie cat mai scurte

Corectitudinea alternantei stricte

- pp. viteze de executie egale pt cele doua procese
- excludere mutuala
  - la inceput, P0 intra in SC, iar P1 asteapta ocupat in ciclul while
  - dupa ce P0 iese din SC, indica ca e randul lui P1 sa intre in SC, si continua cu sectiunea necritica
  - P1 intra in SC si la iesire indica ca e randul lui P0 sa intre in SC

  => doar un proces se afla in SC la un moment dat
- alternanta stricta poate sa nu fie neaparat dezirabila (sau realista)
- oricum, cerinta de progres nu e respectata
  - daca unul dintre procese petrece mult timp in afara SC, celalalt proces e in mod nejustificat impiedicat sa intre SC
- asteptarea limitata e respectata datorita alternantei stricte



Correctness of the Software Solution

- Mutual exclusion is preserved

  P_i enters critical section only if:

  `turn = i`

  and **`turn`** cannot be both 0 and 1 at the same time
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Peterson’s Solution

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section.
  - **`flag[i] =`** ***`true`*** implies that process $\mathbf{P_i}$ is ready!

Algoritmul lui Peterson

- notatie: cele doua procese sunt P_i si P_j, unde j = 1 – i

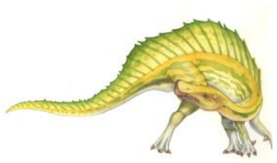
```
const int N = 2;
bool flag[N];
int turn;
void func(int i)
{
        j = 1 – i;
        flag[i] = true;
        turn = j;
        while(flag[j] && (turn == j))
                ;
        sectiune_critica()
        flag[i] = false;
        sectiune_necritica();
}
```

Correctness of Peterson's Solution

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met



Corectitudinea solutiei lui Peterson

- excludere mutuala
  - P_i intra in SC doar daca flag[j] == false sau turn == i
  - pp atat P_i cat si P_j sunt simultan in SC => ambele au executat liniile 7 & 8, deci flag[i] == flag[j] == true
  - cf. celor doua ipoteze de mai sus, P_i si P_j nu puteau executa linia 9 in acelasi timp, pt ca turn nu poate fi si 0 si 1 simultan

  => unul dintre procese, sa zicem P_j trebuie sa fi executat cu succes linia 9 (instructiunea while) in vreme ce celalalt a executat turn = j;
  - dar, in acest moment, flag[j]=true && turn == j si P_i trebuie sa astepte pana cand P_j iese din SC => excluderea mutuala e indeplinita
- progres
- asteptare limitata

Corectitudinea solutiei lui Peterson

- progres
  - P_i nu poate intra in SC doar daca flag[j]=true si turn == j
  - daca P_j nu este interesat sa intre in SC flag[j] == false si P_i poate intra in SC
  - altfel, la iesirea din SC, P_j seteaza flag[j] = false si poate petrece oricat timp in sectiune necritica fara sa impiedice P_i sa intre in SC
- asteptare limitata
  - daca P_j vrea sa intre in SC, seteaza flag[j]=true dar si turn = i
  - pt ca P_i nu schimba valoarea lui turn in linia 9 (instructiunea while), va intra in SC dupa cel mult o intrare a lui P_j in SC

Peterson’s Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson’s Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;
 int x = 0;
```

- Thread 1 performs

```
while (!flag)
   ;
 print x
```

- Thread 2 performs

```
  x = 100;
 flag = true
```

- What is the expected output?

  100

Modern Architecture Example (Cont.)

- However, since the variables `flag` and `x` are independent of each other, the instructions:

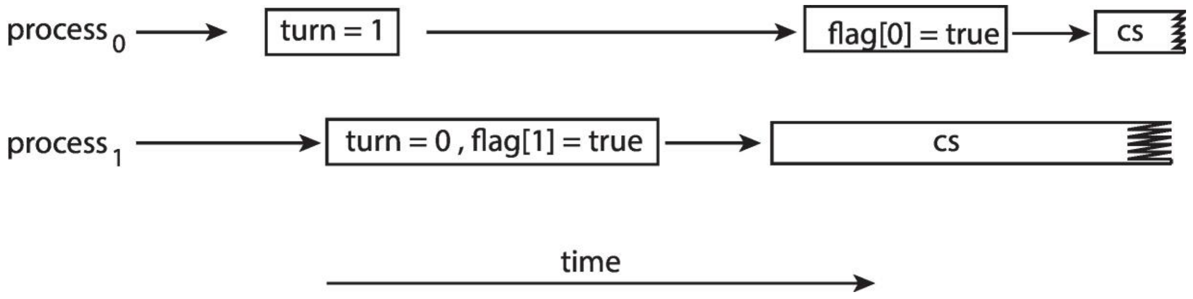
```
flag = true;
 x = 100;
```

 for Thread 2 may be reordered

- If this occurs, the output may be 0!

Peterson’s Solution Revisited

- The effects of instruction reordering in Peterson’s Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson’s solution will work correctly on modern computer architecture we must use **Memory Barrier**.

Memory Barrier

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Memory Barrier Example

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
 while (!flag)
   memory_barrier();
print x
```

- Thread 2 now performs

```
 x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

1. Hardware instructions
2. Atomic variables

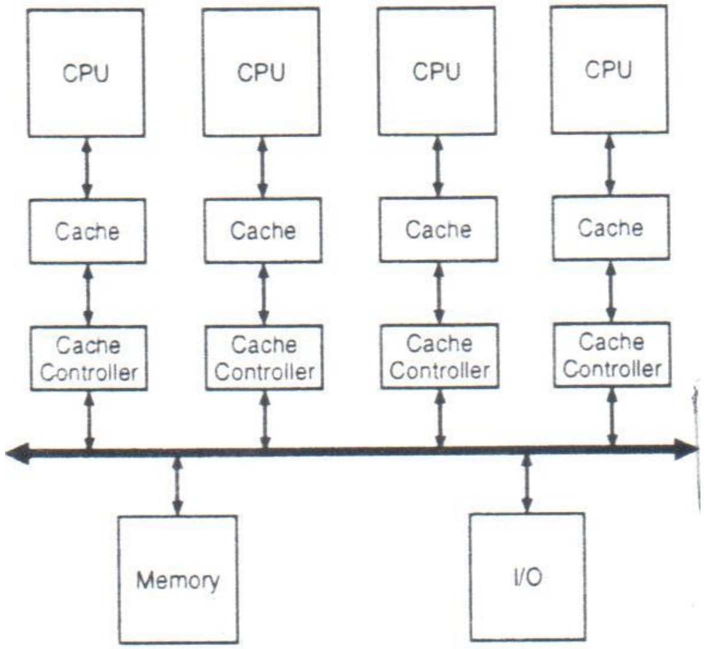
Hardware Instructions

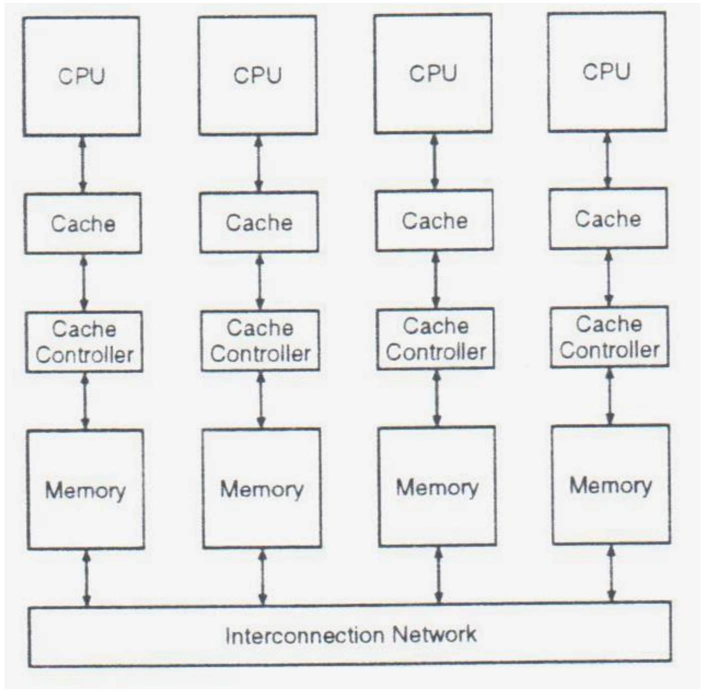
- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction

Sisteme multiprocesor (MP)

- taxonomie arhitecturi multiprocesor
  - Flynn: SISD, SIMD, MISD, MIMD
  - MIMD sunt referite uzual ca sisteme multiprocesor
- felul in care se interconecteaza procesoarele individuale in sistemele multiprocesor determina in general si tipul de acces la memorie
  - UMA, NUMA, NORMA
- in sistemele UMA/NUMA procesoarele partajeaza date protejat prin variabile partajate de memorie accesate sincronizat (cu ajutorul lock-urilor, de pilda)
- in sistemele NORMA accesul coordonat la date se face prin message passing
- vom restrange discutia la multiprocesoare interconectate printr-o singura magistrala

MP cu memorie partajata





Multiprocesoare cu o singura magistrala

- magistrala unica reprezinta principala limitare pt. cresterea nr. de CPU
- folosirea cache-urilor reduce traficul pe magistrala si permite cresterea nr de procesoare in arhitecturile UMA
- existenta cache-urilor in general (si pt cc-NUMA) => copii multiple ale aceleiasi locatii de memorie => potential de inconsistenta daca un CPU modifica datele (locale)
- accesul uncached la datele partajate nu e indicat
  - scade viteza accesului la date
  - creste traficul pe magistrala ceea ce incetineste si accesul celorlalte procesoare inclusiv la datele nepartajate
- ideal, cand un procesor citeste date partajate pot exista copii multiple si in alte cache-uri, dar la scriere e nevoie de acces exclusiv la date urmat de invalidarea celorlalte copii existente

Multiprocesoare cu o singura magistrala

- solutia: protocoale de mentinere a coerentei cache-urilor locale
- ex de protocol uzual: *snoop coherency*
  - controlerele cache-urilor monitorizeaza magistrala pt traficul care afecteaza datele din cache-urile lor
  - *read miss*: controlerul localizeaza o copie actualizata a datelor (posibil aflata in cache-ul altui CPU)
  - *write*: exista doua protocoale posibile, *write-invalidate* si *write-update*
  - *write-invalidate* invalideaza toate copiile celorlalte procesoare inainte de a scrie datele in cache-ul local (eg. Intel MESI)
  - *write-update* scrie datele modificate pe magistrala printr-o operatie de broadcast
    - controlerele de cache isi pot actualiza copiile locale ale datelor la valoarea modificata

Suport HW pentru sincronizare

- multe procecesoare ofera suport HW pt implementarea sectiunilor critice
- uniprocesoare – pot dezactiva intreruperile
  - codul current se executa fara a fi interrupt (preempted)
  - ineficient in sisteme multiprocesor
- instructiuni HW atomice
  - instructiuni procesor speciale executate *in mod atomic (neintrerupt)*
  - fie testeaza si modifica (*test-and-modify)* continutul unui cuvant de memorie, **Test-and-Set (TAS)**
  - fie schimba (*swap)* continutul a doua cuvinte de memorie (sau mai multe), **Compare-and-Swap** (**CAS**, cu varianta **CAS2**)

Test-and-set (TAS)

- definitie

```
int tas(int *lock)
  {
          int rv = *lock;
          *lock = 1;
          return rv:
  }
```

- proprietati
  - executie atomica
  - intoarce valoarea originala a parametrului pasat functiei
  - seteaza noua valoarea a parametrului pe 1

Spinlock-uri cu TAS

- locks (lacat, zavor), concept (abstractie) de nivel inalt pt. protectia sectiunilor critice
- ofera doua operatii: *acquire* (apelata la intrarea in sectiune critica) si respectiv *release* (apelata la iesirea din sectiune critica)
- cand un proces/thread detine lock-ul, celelalte sunt in busy waiting (“spinning in the while loop”), a.k.a *spinlocks*
- pt a se reduce busy waiting, sectiunile critice implementate cu spinlocks trebuie sa fie foarte scurte

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

Instructiuni atomice multiprocesor

- se folosesc instructiuni atomice de tip TAS/CAS
- TAS in sisteme multiprocesor
  - cand un CPU executa TAS, arbitrul de magistrala da drept de folosinta exclusiva a magistralei procesorului respectiv pe durata executiei instructiunii
  - toate celelalte surse generatoare de accese de memorie sunt blocate pe durata TAS
  - se executa ciclul read-modify-write
  - la final, se deblocheaza magistrala pt uzul altor procesoare

Consecinte ciclu read-modify-write

- intr-un astfel de ciclu se citeste si se scrie o valoare in mod atomic, iar datele nu sunt cached

=> operatie mai costisitoare decat operatiile de citire/scriere obisnuite

=> magistrala e ocupata mai mult timp

=> operatiile TAS afecteaza semnificativ rata de transfer pe magistrala

- alte consecinte privesc implementarea spinlock-urilor in sisteme multiprocesor

Spinlock-uri TAS MP

```
void acquire(int *lock_ptr)
{
        disable_interrupts();
        while(tas(lock_ptr))
                ;
}
void release(int *lock_ptr)
{
        *lock_ptr = 0;
        enable_interrupts();
}
```

Observatii

- dezactivarea intreruperilor in *acquire* impiedica pierderea procesorului pe durata detinerii lock-ului
- pierderea procesorului ar insemna ca procese/thread-uri de pe alte CPU nu pot intra in sectiune critica din cauza unui proces/thread care nu ruleaza !
- pe durata detinerii unui lock, celelalte procesoare executa TAS in bucla si consuma inutil largimea de banda a magistralei afectand procesoare care nu sunt implicate in accesul la sectiunea critica !
- la *release*, procesorul care detine lock-ul concureaza cu celelalte procesoare pt accesul la magistrala inainte de a putea ceda lock-ul
- solutie: *test-and-test-and-set* (TATAS)

TATAS

- traficul pe magistrala se poate reduce daca procesoarele cicleaza in *acquire* pe o copie locala a spinlock-ului
- pt ca *release* e in fond o operatie de scriere a spinlock-ului, protocolul de coerenta snoop detecteaza scrierea si invalideaza copiile locale ale celorlalte procesoare (in cazul protocolului *write-invalidate*)
- cand celelalte procesoare acceseaza din nou spinlock-ul => cache miss
- la cache miss copia locala se va actualiza si va reflecta starea de lock free
- procesorul incearca din nou sa execute operatia TAS sperand ca acum lock-ul e probabil free

Spinlock-uri TATAS

```
void acquire(int *lock_ptr)
{
        disable_interrupts();
        while(*lock_ptr || tas(lock_ptr))
                ;
}
void release(int *lock_ptr)
{
        *lock_ptr = 0;
        enable_interrupts();
}
```

Obs: codul se bazeaza pe evaluarea scurtcircuitata a conditiilor in C

TATAS vs TAS

- pt sectiuni critice scurte si protocol *write-invalidate*, costurile sunt asemanatoare
  - dupa ce un CPU elibereaza lock-ul si altul il obtine, dureaza pana cand celelalte procesoare ajung sa cicleze pe copia locala a lock-ului
  - in tot acest timp, magistrala e saturata cu trafic de invalidare, read miss si TAS
- secventa tipica de evenimente
  - eliberare lock => invalidare copii din cache-urile altor CPU
  - invalidarile genereaza read miss pt toate CPU care cicleaza, toate citesc noua valoare a lock-ului de pe bus si ordinea e seriala !
  - primul CPU care obtine noua valoare a lock-ului, executa TAS si obtine lock-ul
  - restul CPUs primesc noua valoare, executa TAS si esueaza
  - fiecare TAS invalideaza copiile locale din cache-uri si forteaza un read miss

TATAS cu backoff

- competitia pt. magistrala partajata seamana cu protocolul de transmisie a datelor pe Ethernet (CSMA/CD)
  - analogia nu e stricta: pe Ethernet toti transmitatorii simultani esueaza, la TAS pe MP arbitrul de magistrala garanteaza un castigator
- analogia sugereaza insa scheme de backoff pt. reducerea competitiei la magistrala a procesoarelor care cicleaza
- metoda introduce o intarziere (delay) inainte de a incerca din nou operatia TAS
- intarzierea poate fi statica sau dinamica
- intarzierea statica:
  - fiecare CPU are un slot
  - merge bine pentru multe CPU-uri
  - intarzie nejustificat un singur CPU chiar daca lock-ul e liber

TATAS cu backoff (cont.)

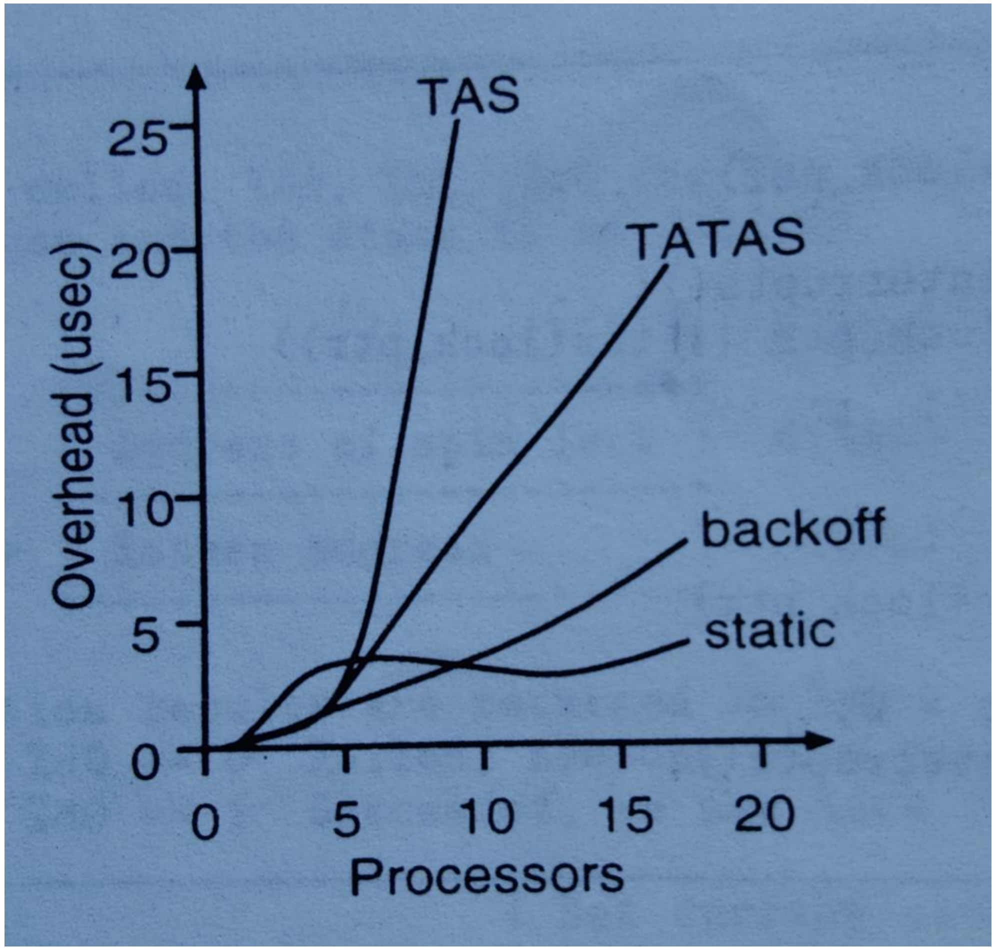
- intarzierea dinamica:
  - la inceput, toate CPU-urile aleg o intarziere mica => coliziuni
  - dupa detectarea coliziunilor, fiecare CPU mareste intarzierea
  - overhead-ul intarzierilor mici de la inceput face metoda mai costisitoare decat alocarea statica a intarzierilor
- Obs: folosirea protocolului *write-update* poate imbunatati performanta TATAS prin reducerea traficului pe magistrala
  - cand un lock e eliberat, noua sa valoare poate fi transmisa prin broadcast
  - fiecare CPU care monitorizeaza magistrala isi poate astfel actualiza copia locala fara a mai genera read miss

Spinlock-uri TATAS cu backoff

```
void acquire(int *lock_ptr)
{
        disable_interrupts();
        while(*lock_ptr || tas(lock_ptr))
        {
                while(*lock_ptr)
                        ;
                delay();
        }
}
```

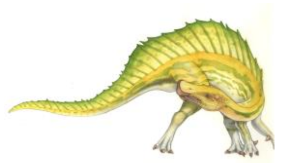
```
void release(int *lock_ptr)
{
        *lock_ptr = 0;
        enable_interrupts();
}
```

Performanta TAS



Dezavantaje instructiuni atomice

- instructiunile atomice simplifica IPC pe multiprocesoare, dar au si dezavantaje
  - pot avea performanta redusa daca sunt folosite neglijent
  - daca un proces/thread pierde CPU sau e fortat sa astepte o perioada lunga (eg, page fault) in timp ce detine lock-ul, celelalte procese nu pot avansa pana cand detinatorul lock-ului revine pe CPU si elibereaza lock-ul
  - daca un proces/thread se termina anormal in timp ce detine un lock, nici un alt proces/thread nu poate intra in sectiune critica
  - inversarea prioritatilor – cand un proces/thread cu prioritate mica detine lock-ul si impiedica un proces cu prioritate marea sa intre in sectiune critica
- solutie: sincronizare *wait-free* (*lock-free*), Herlihy 1991



Sincronizare wait-free (lock-free)

- idee centrala: se incearca executia operatiei, dar se lasa datele intr-o stare consistenta daca operatia esueaza
  - analogie cu tranzactiile din baze de date
  - model de control optimist al concurentei
- se pot folosi instructiuni HW de tipul CAS/CAS2 sau LL/SC
- model Herlihy:
  - sistem cu *n* procese secventiale care comunica prin memorie partajata
  - *obiect concurent* = ({tip+valori}, set operatii, specificatie secventiala)
  - *obiect concurent neblocant* – procesul care executa una dintre operatiile sale trebuie o termine intr-un nr finit de pasi
  - *obiect concurent wait-free* – fiecare proces din sistem trebuie sa termine operatia intr-un nr finit de pasi

Sincronizare wait-free (cont.)

- conditia de operare *nonblocanta* - unele procese/threaduri vor face intotdeauna progres indiferent de intarzierile sau opririle fortate ale altor procese/threaduri
- conditia de operare *wait-free* e mai puternica => toate procesele/threadurile **care nu sunt oprite** progreseaza indiferent de terminarile anormale sau intarzierile altor procese/threaduri
- ambele conditii **exclud** folosirea sectiunilor critice ca metoda de implementare
  - un proces/thread care se termina anormal (cu eroare) in mijlocul unei sectiuni critice va bloca pt totdeauna celelalte procese care asteapta sa intre in sectiunea critica
- **rezultat teoretic**: *este imposibil sa se construiasca implementari neblocante sau wait-free ale tipurilor de date simple folosind operatii de citire/scriere, TAS, fetch-and-add, swap de memorie cu registre* (eg, instructiunea Intel *xchg*)

Semantica compare-and-swap (CAS)

- definitie

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

- proprietati
  - executie atomica
  - intoarce valoarea originala a parametrului **`value`**
  - seteaza variabila **`value`** la valoarea parametrului **`new_value`** doar daca e indeplinita conditia **`*value == expected`**, adica, schimbul de valori se face doar daca e indeplinita conditia

Incrementare atomica wait-free cu CAS

```
1       void increment(atomic_int *v)
2       {
3               int temp;
4               do {
5                       temp = *v;
6               }while (temp != (compare_and_swap(v,temp,temp+1));
7       }
```

Scenariu de executie:

- pp initial v = 10
- pp. proces de prioritate mica P_l executa codul primul
- P_l pierde CPU intre liniile 5 si 6, moment in care un proces cu prioritate mare P_h incepe executia
- starea P_l este v=10, temp=10, iar P_h incheie cu succes incrementarea => v = 11
- cand P_l se reia, **cas(11,10,11)** esueaza si se re-executa linia 5 => temp = v = 11 si **cas(11, 11, 12)** reuseste = > v = 12

Obs: spre deosebire de TAS care modifica operandul intotdeauna, CAS/CAS2 modifica operandul *doar daca reuseste comparatia* !

Suport HW RISC pentru sectiuni critice

- TAS, CAS specifice procesoarelor CISC, dar secvente atomice de tip *read-modify-write* nu se pot implementa pe procesoare RISC (arhitecturi load/store)
- operatii speciale RISC: Load Linked (LL) si Store Conditional (SC)
- LL incarca o variabila din memorie intr-un registru CPU si apoi verifica activ daca variabila din memorie este modificata de alte procesoare
- SC verifica daca au existat modificari ale variabilei de memorie de la ultimul LL
  - daca nu, valorea registrului se stocheaza in memorie cu success (variabila din memorie este modificata) iar continutul registrului este setat pe 1
  - daca da, stocarea esueaza (variabila de memorie ramane nemodificata) iar valoarea registrului se seteaza pe 0

Incrementare atomica wait-free cu LL/SC

- in fapt, o forma elementara de memorie tranzactionala (zona de memorie restransa la un byte/word)
- functioneaza cf. principiilor de control optimist al concurentei din baze de date
- simuleaza executia unei tranzactii din baze de date cf principiilor ACID (Atomicity, Consistency, Isolation, Durability)

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

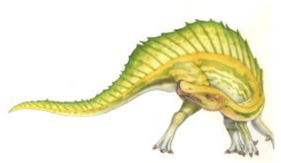
Proprietati ACID incrementare LL/SC

- *atomicitate* - modificarile (incrementarea counter-ului) sunt executate ca si cand operatia ar fi atomica, i.e. fie toate modificarile sunt executate, fie niciuna
- *consistenta* - datele (*count*) sunt intr-o stare consistenta cand tranzactia incepe si cand se termina, i.e. *count* e incrementat corect, chiar daca se executa mai multe tranzactii simultan, eventual intretesut
- *izolare* - starea intermediara a tranzactiei (apelul procedurii si valorile registrelor) e invizibila altor tranzactii (altor apeluri ale aceleiasi proceduri)
  - tranzactiile concurente (apelurile concurente ale procedurii *increment*) par a fi serializate
- *durabilitate* – dupa incheierea cu succes a tranzactiei (apelul procedurii *increment*), datele sunt salvate in memoria principala RAM si modificarea e finala
  - analogia cu bazele de date se opreste aici, stocarea in RAM nu e persistenta

Comparatie CAS vs. LL/SC

- daca au aparut modificari ale counter-ului, SC va esua garantat, chiar daca valoarea initial citita de LL a fost restaurata
  - ex: problema ABA, P1 si P2 partajeaza o variabila de memorie
  - P1 citeste A si pierde procesorul, P2 scrie B si apoi A, P1 revine pe procesor si citeste A
- daca se incearca aceeasi secventa de operatii cu CAS, adica
  - se citeste valoarea counter-ului
  - apoi se executa CAS
  - daca vechea valoare a fost restaurata, nu se vor detecta modificarile intermediare aparute intre operatia de read si cea de CAS

  => semantica LL/SC e mai puternica decat CAS
- atat CAS cat si LL/SC se pot folosi pentru implementarea sincronizarii *wait-free*



Memorie tranzactionala

- software transactional memory, Herlihy & Moss 1993
- am vazut deja ex. LL/SC & CAS
- exploateaza ideea de tranzactii din baze de date (v. proprietati ACID) si controlul optimist al concurentei
- un thread termina modificarile operate pe o zona de memorie partajata fara a se coordona cu alte threaduri
- se inregistreaza fiecare operatie de citire/scriere intr-un log
- la finalul tranzactiei se incearca operatia de *commit*
  - daca reuseste (i.e., nici o alta tranzactie nu a apucat sa faca un *commit* reusit intre timp), tranzactia e validata si modificarile devin permanente
  - daca esueaza, tranzactia e abandonata (*aborted*) si se re-executa de la inceput pana reuseste (*transaction roll-back*)
- beneficiu major: grad crescut de concurenta, thread-urile nu au nevoie sa se sincronizeze prin lock-uri la accesul memoriei partajate

Suport HW pt. memorie tranzactionala

- ex. Intel TSX (Transactional Synchronization Extension), extensie ISA x86, microarhitectura Haswell (2013)
- accesibila prin intermediul a doua interfete
  - HLE (Hardware Lock Elision) – backward compatibility
  - RTM (Restricted Transactional Memory)
- HLE adauga doua prefixuri de instr. XACQUIRE & XRELEASE
  - se pot folosi doar pt anumite instructiuni care trebuie prefixate explicit cu LOCK
  - permite executia optimista a sectiunii critice sarind scrierea lock-ului, a.i. acesta apare liber pt alte threaduri
  - o tranzactie esuata determina reluarea operatiei de la instructiunea prefixata cu XACQUIRE, dar instructiunea se va reexecuta ca si cand nu a fost prefixata cu XACQUIRE

Suport HW pt. memorie tranzactionala

- RTM adauga la ISA trei noi instructiuni
  - XBEGIN – marcheaza inceputul zonei de memorie tranzactionala
  - XEND - marcheaza sfarsitul zonei de memorie tranzactionala
  - XABORT – abandoneaza explicit o tranzactie
  - esecul tranzactiei redirecteaza executia catre codul specificat de instr. XBEGIN, iar codul de eroare e stocat in registrul EAX
- instructiunea XTEST permite testarea starii procesorului (este sau nu in mijlocul executiei unei regiuni de memorie tranzactionala)

RTM lock elision in glibc

```
void elided_lock_wrapper(lock) {
        if(_xbegin() == _XBEGIN_STARTED) { // start tranzactie
                if(lock  e liber)
                        return;    // executa sectiunea critica in tranzactie
                _xabort(0xff);     // abandoneaza tranzactia
        }
        obtine lock
}
void elided_unlock_wrapper(lock) {
        if (lock e liber)
                _xend();           // comite tranzactia
        else
                elibereaza lock;
}
```

The test_and_set Instruction

- Definition

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

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to **`true`**

Solution Using 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);
```

- Does it solve the critical-section problem?

The 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;
  }
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **`value`**
  - Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer **`lock`** initialized to 0;
- Solution:

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

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

Bounded-waiting with compare-and-swap

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The Command:

    ```
    increment(&sequence);
    ```

    ensures **`sequence`** is incremented without interruption:

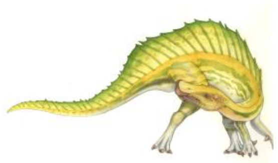
Atomic Variables

- The **`increment()`** function can be implemented as follows:

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

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
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**



Solution to CS Problem Using Mutex Locks

```
while (true) {
      acquire lock

          critical section

      release lock

remainder section
}
```

Sleep/wakeup

- spinlock-urile sufera de busy-waiting si ca atare de problemele discutate anterior
- busy-waiting-ul se poate evita cu ajutorul unei primitive **sleep**
- **sleep** blocheaza procesele care nu au dreptul sa intre in sectiune critica (pierd procesorul)
- cand procesul aflat in sectiune critica paraseste sectiunea critica, apeleaza primitiva **wakeup** care trezeste toate procesele blocate in sleep
- problema producator-consumator implementata cu **sleep/wakeup**

Producator-consumator

```
count = 0
N = dim buffer
producator()
{
    while(1)
    {
            produce_item();
            if(count == N) sleep();
            enter_item();
            count++;
            if(count == 1)
                wakeup(consumator)
    }
}
```

```
consumator()
{
    while(1)
    {
            if(count == 0) sleep();
            remove_item();
            count--;
            if(count == N - 1)
                wakeup(producator)
            consuma_item();
    }
}
```

Race condition solutie sleep/wakeup

- pp buferul e gol, consumatorul a citit *count* == 0 si inainte sa execute testul schedulerul da controlul producatorului
- producatorul produce un obiect, observa ca buferul era gol, crede ca procesul consumator executa **sleep** si apeleaza **wakeup** pt a-l notifica
- cand revine pe CPU, consumatorul executa testul si apoi **sleep** => notificarea **wakeup** s-a pierdut !
- pt ca procesul consumator nu a apucat sa consume obiectul produs, producatorul continua sa produca obiecte pana umple buferul complet si e nevoit sa apeleze **sleep**
- in acest moment, ambele procese sunt blocate => **deadlock !**
- problema esentiala: accesul nesincronizat la variabila *count*
- solutie posibila: **wakeup** seteaza un waiting bit pe care consumatorul il citeste si nu intra in **sleep**
    - limitare neplacuta: in general e nevoie de un nr arbitar de waiting bits => e necesara o primitiva speciala (semafoare)

Semafoare

- structura de date speciala (Dijkstra 1965)
  - are contor care numara **wakeup**-urile
  - are coada de procese blocate in **sleep**
  - **down**
    - operatie care decrementeaza **atomic** contorul daca e > 0 si permite procesului sa continue
    - daca contorul e zero, blocheaza procesul si il pune in coada
  - **up** – verifica daca exista procese blocate in coada
    - daca da, alege unul dintre ele si il deblocheaza
    - daca nu, incrementeaza **atomic** contorul
- un semafor cu contor initial 1 este de fapt un mutex (semafor binar)
- **up/down** generalizeaza **sleep/wakeup** DAR sunt operatii **atomice** !
- obs: **up** nu blocheaza niciodata procese !

Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "`mutex`" initialized to 1

```
down(mutex);
    CS
up(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    up(synch);
P2:
    down(synch);
    S2;
```

Producator-consumator cu semafoare

```
item_t buffer[N]
semaphore mutex = {1}, space = {N},
items = {0};
void producer(void)
{
        item_t item;
        while(true) {
            produce(&item);
            down(space);
            down(mutex);
            put(buffer, &item);
            up(mutex);
            up(items);
        }
}
```

```
void consumer(void)
{
        item_t item;
        while(true) {
            down(items);
            down(mutex);
            get(buffer, &item);
            up(mutex);
            up(space);
            consume(&item);
        }
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute the **`down()`** and **`up()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`down`** and **`up`** 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

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Implementation with no Busy waiting (Cont.)

- Waiting queue

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

Implementation with no Busy waiting (Cont.)

```
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);
   }
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `up(mutex)` …. `down(mutex)`
  - `down(mutex)` … `down(mutex)`
  - Omitting of `down (mutex)` and/or `up (mutex)`
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

Ex. greseli producator-consumator

- producatorul inverseaza secventa de accesare a semafoarelor

```
        down(mutex);
        down(space);
```

- daca buferul e plin, producatorul se blocheaza in operatia down(space), iar mutex = 0
- consumatorul porneste si executa

```
        down(items);
        down(mutex)
```

=> Deadlock ! (mutex = 0)

- concluzie: e nevoie de mecanisme de sincronizare de nivel inalt asistate de compilator !

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

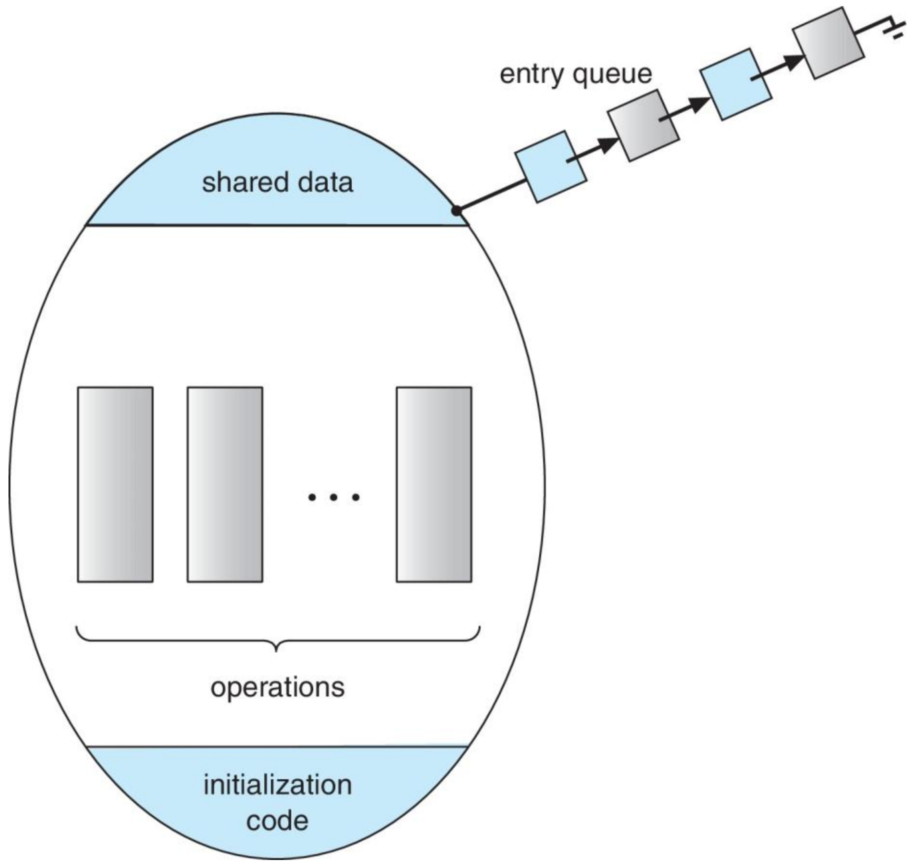
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure P2 (…) { …. }

  procedure Pn (…) {……}

   initialization code (…) { … }
}
```

Schematic view of a Monitor



Asistenta compilatorului

- procedurile monitorului sunt instrumentate de compilator sa execute o secventa speciala de apelare
- cand un proces apeleaza o procedura din monitor
  - primele instructiuni ale procedurii verifica daca exista alt proces activ in interiorul monitorului
  - daca da, procesul e suspendat pana cand celalalt proces paraseste monitorul
  - daca nu, procesul apelant poate intra in monitor
  - implementarea se face in mod uzual folosind un semafor
- avantaj: sectiunile critice se implementeaza ca proceduri de monitor, iar compilatorul genereaza automat cod de excludere mutuala

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex
mutex = 1
```

- Each procedure ***P*** is replaced by

```
down(mutex);
     …
  body of P;
     …
up(mutex);
```

- Mutual exclusion within a monitor is ensured

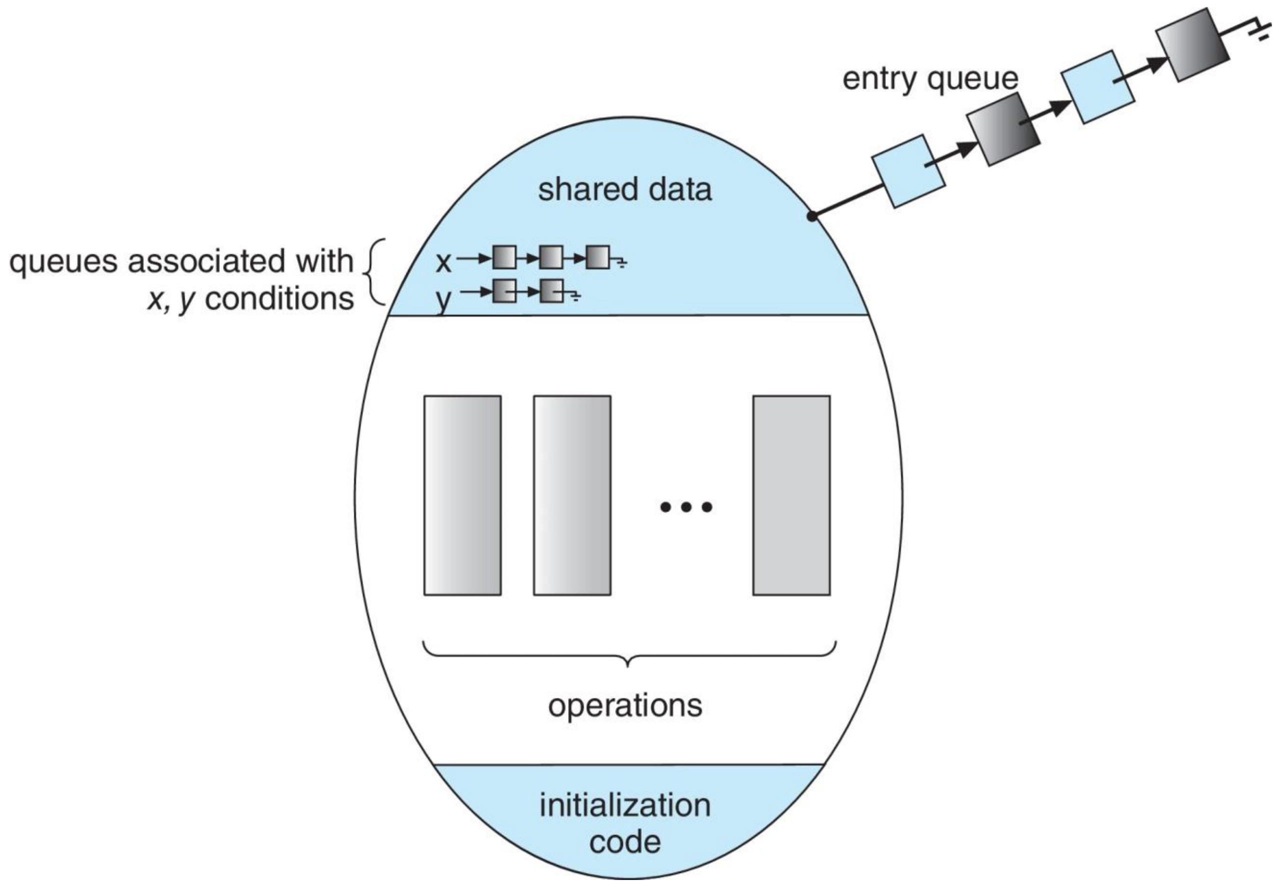
Variabile conditie

- problema: ce se intampla daca un proces aflat in interiorul monitorului **trebuie** sa se blocheze? (eg., situatia producatorului cand buferul e plin)
- solutie: **variabile conditie**
- suporta doua operatii
  - **wait** – blocheaza procesul apelant si elibereaza monitorul pt ca alte procese blocate la intrarea in monitor sa poata accesa monitorul
  - **signal** – trezeste procesul care a apelat anterior *wait* pe aceasta variabila conditie
- problema **signal** – daca trezeste un proces care apelase anterior **wait**, vor exista simultan doua procese in interiorul monitorului
- solutii
  - Hoare – procesul care executa **signal** e suspendat, si celalalt proces este lasat sa ruleze in monitor
  - Hansen – **signal** nu se poate executa decat ca ultima operatie a unei proceduri de monitor (i.e., procesul care cheama **signal** paraseste imediat monitorul dupa executia operatiei)
  - uzual, se foloseste solutia Hansen pt. simplitatea implementarii

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable

Monitor with Condition Variables



Observatii

- variabilele conditie nu contorizeaza “semnalele”, i.e. o operatie **signal** pe o variabila conditie pe care nu asteapta nimeni se pierde
  - **wait** trebuie executat inainte de **signal**
- **wait/signal** aproximeaza comportamentul **sleep/wakeup**, DAR race condition-ul **sleep/wakeup** e inlaturat de excluderea mutuala automata a accesului in monitor
- 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

Usage of Condition Variable Example

- Consider P_1 and P_2 that that need to execute two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a monitor with two procedures F_1 and F_2 that are invoked by P_1 and P_2 respectively
  - One condition variable “x” initialized to 0
  - One Boolean variable “done”
  - **F1:**

```
        S1;
        done = true;
        x.signal();
```

  - **F2:**

```
        if done = false
            x.wait()
        S2;
```

Echivalenta primitivelor de sincronizare

- event counters, sequencers, path expressions, serializers, toate sunt semantic echivalente
- exemplu practice: echivalenta monitoarelor cu semafoarele
- pp. sistemul de operare ofera semafoare
- dezvoltatorul de compilatoare scrie urmatoarele rutine

```
semaphore_t mutex = {1};
void enter_monitor(void)    { down(mutex); }
void leave_normally(void)   { up(mutex); }

void leave_with_signal(semaphore_t c)  // c = 0 initial
{   up(c); // iese cu mutex = 0, i.e. nici un alt proces nu poate intra
             // (signal e ultima instructiune din procedura)
}
void wait(semaphore_t c)   { up(mutex); down(c); }
```

Implementare semafoare cu monitoare

- exemplu in Java

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

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of processes waiting
                       inside the monitor
```

- Each function ***P*** will be replaced by

```
down(mutex);
     …
   body of P;
     …
if (next_count > 0)
 up(next)
else
 up(mutex);
```

- Mutual exclusion within a monitor is ensured

Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

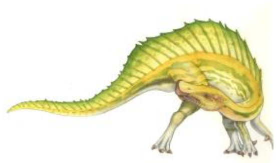
- The operation **`x.wait()`** can be implemented as:

```
x_count++;
if (next_count > 0)
    up(next);
else
   up(mutex);
down(x_sem);
x_count--;
```

Implementation (Cont.)

- The operation **`x.signal()`** can be implemented as:

```
if (x_count > 0) {
   next_count++;
   up(x_sem);
   down(next);
   next_count--;
}
```



Resuming Processes within a Monitor

- If several processes queued on condition variable **`x`**, and **`x.signal()`** is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the **conditional-wait** construct of the form

```
x.wait(c)
```

where:

- **`c`** is an integer (called the priority number)
- The process with lowest number (highest priority) is scheduled next

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource

```
R.acquire(t);
        ...
   access the resurce;
        ...

R.release;
```

- Where R is an instance of type **ResourceAllocator**

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource
- The process with the shortest time is allocated the resource first
- Let R is an instance of type **`ResourceAllocator`** (next slide)
- Access to **`ResourceAllocator`** is done via:

```
R.acquire(t);
      ...
  access the resurce;
      ...
R.release;
```

- Where **`t`** is the maximum time a process plans to use the resource

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = true;
   }
   void release() {
            busy = false;
            x.signal();
   }
   initialization code() {
    busy = false;
   }
}
```

Single Resource Monitor (Cont.)

- Usage:

```
acquire
 ...
release
```

- Incorrect use of monitor operations
  - **`release()`** … **`acquire()`**
  - **`acquire()`** … **`acquire())`**
  - Omitting of **`acquire()`** and/or **`release()`**

Lock-uri adaptive

- primitive de sincronizare Solaris: lock-uri adaptive (adaptive locks), variabile conditie, semafoare, reader-writer locks, turnstiles
- lock-uri adaptive
  - comportament polimorf spinlock-semafor
  - pt sectiuni critice scurte (cateva sute de instructiuni)
    - daca un thread vrea acces la date protejate de un lock detinut de un alt thread care ruleaza pe alt CPU, lock-ul se comporta ca un spinlock
  - daca lock-ul e detinut de un thread care nu ruleaza momentan pe nici un procesor, threadul care incearca sa obtina lock-ul va fi pus in stare dormanta (sleep), i.e., lock-ul se comporta ca un semafor
  - pe un sistem uniprocesor, cand un thread incearca sa obtina un lock detinut de alt thread, comportamentul lock-ului e intotdeauna de tip semafor

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

Liveness

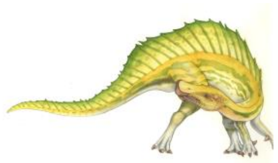
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

```
        P0                  P1
    down(S);            down(Q);
    down(Q);            down(S);
      ...                 ...
    up(S);              up(Q);
    up(Q);              up(S);
```

- Consider if P_0 executes down(S) and P_1 down(Q). When P_0 executes down(Q), it must wait until P_1 executes up(Q)
- However, P_1 is waiting until P_0 execute up(S).
- Since these up() operations will never be executed, P_0 and P_1 are **deadlocked**.

Conditii necesare pt. deadlock

- Coffman et al., 1971
- (1) **excludere mutuala (mutual exclusion)** – doar un proces/thread utilizeaza o resursa la un moment dat
- (2) **hold & wait** – procesul/threadul detine o resursa si asteapta sa obtina alte resurse detinute de alte procese/threaduri
- (3) **fara preemptiune (no preemption)** – resursele detinute de un proces/thread nu pot fi confiscate fortat de la un proces/thread care le detine, ci doar cedate voluntar la sfarsitul taskului
- (4) **asteptare circulara (circular wait)** – un set de procese/threaduri aflate intr-un lant de asteptare (P_i asteapta o resursa detinuta de P_{i+1})
- producerea unui deadlock presupune ca toate cele 4 conditii sa fie respectate !
- obs: (4) => (2), adica nu toate conditiile sunt independente unele de celelalte



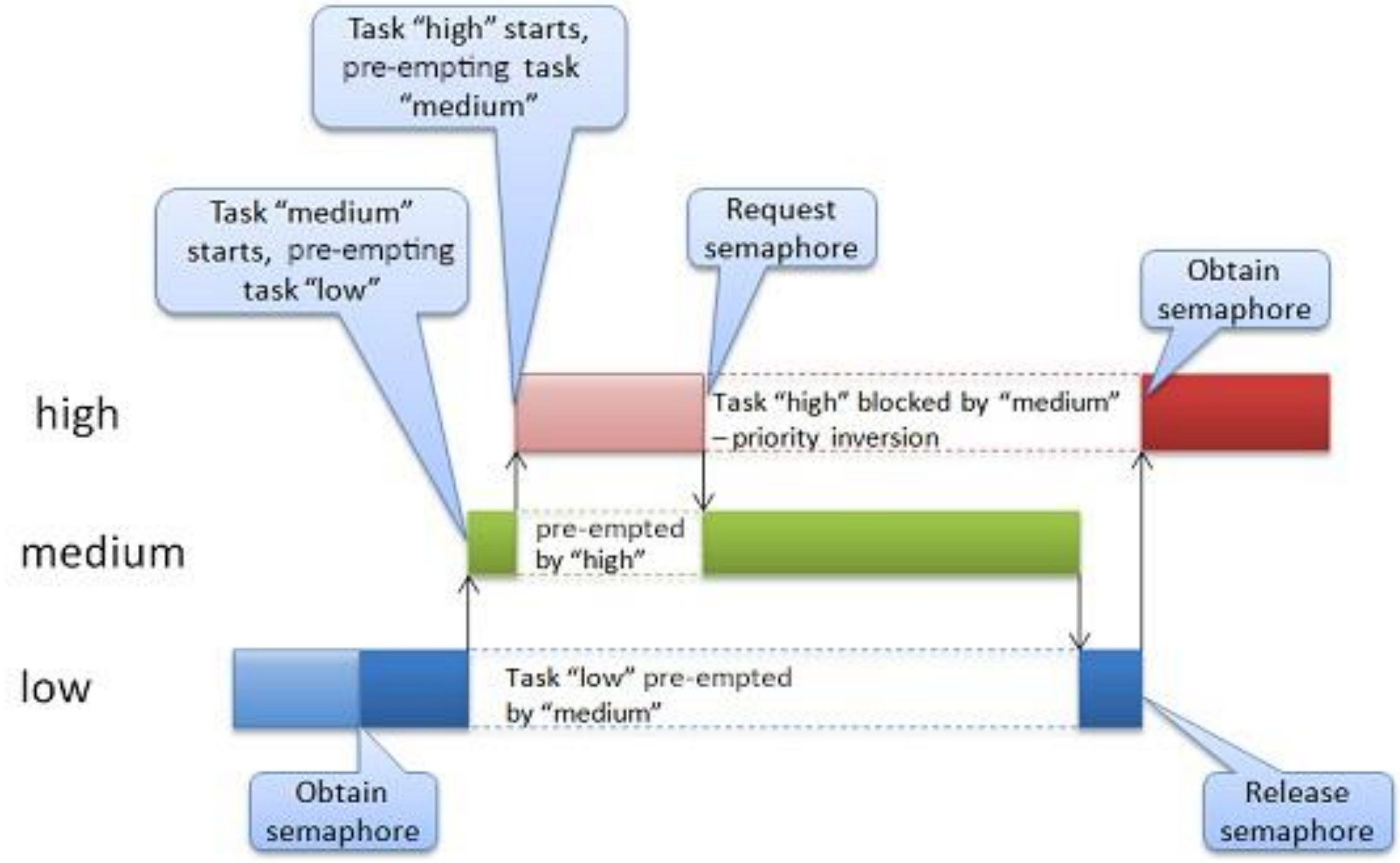
Liveness

- Other forms of deadlock:
- **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**

Incidentul Mars Pathfinder

- aplicatiile roverului rulau pe VxWorks (sistem de operare de timp real)
- taskurile erau planificate in functie de prioritati
  - task de mica prioritate care aduna date meteorologice, accesa magistrala folosind un mutex
  - task de gestiune a magistralei de prioritate mare, accesa magistrala folosind acelasi mutex
  - task de comunicare de lunga durata, cu prioritate medie, impiedica taskul meteorologic sa ruleze
- problema: un timer de tip watchdog reseta periodic sistemul observand ca taskul de gestiune a magistralei nu a mai rulat de mult si concluzionand ca s-a defectat

Inversiunea de prioritati


Task "high" starts, pre-empting task "medium"
Task "medium" starts, pre-empting task "low"
Request semaphore
Obtain semaphore
high
Task "high" blocked by "medium" – priority inversion
medium
pre-empted by "high"
low
Task "low" pre-empted by "medium"
Obtain semaphore
Release semaphore

Mostenirea prioritatii

- solutia problemei: *priority inheritance*
- taskul care detine mutex-ul (semaforul) mosteneste prioritatea taskului cu prioritate mare pe durata sectiunii critice
- astfel, taskul de prioritate mica (eg., taskul meteorologic) primeste procesorul in locul taskului de prioritate medie (eg., taskul de comunicare)
- in aceste conditii, taskul de prioritate mica care si-a marit temporar prioritatea
  - termina sectiunea critica fara sa piarda procesorul
  - elibereaza mutex-ul (resursa partajata, eg., magistrala)
  - revine la prioritatea mica

End of Chapter 6