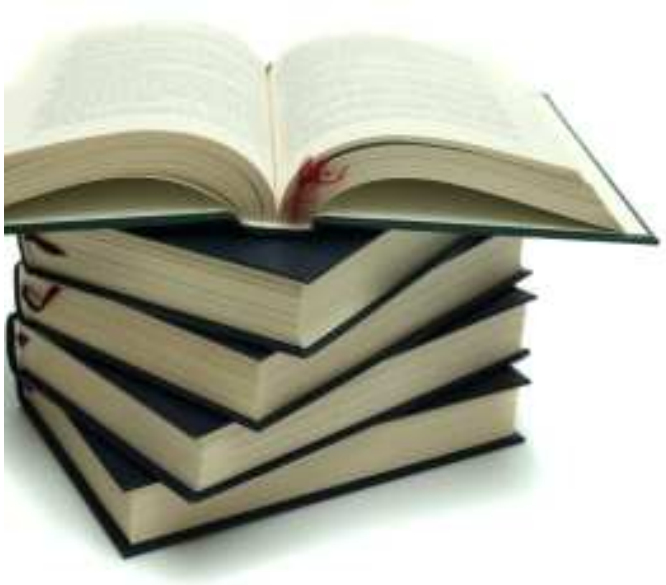
page crunch

Cursul 5



5

Sincronizarea proceselor

25 martie 2009

- OSC
  - Capitolul 6 – Process Synchronization
    - Secțiunile 6.1-6.5, 6.6.1, 6.6.2, 6.8.2-6.8.4
  - Capitolul 7 - Deadlocks
- MOS
  - Capitolul 2 – Processes and Threads
    - Secțiunile 2.3.1–2.3.6, 2.3.9, 2.4.2
  - Capitolul 3 - Deadlocks
- Little Book of Semaphores
  - Capitolele 1, 2, 3
  - Capitolul 4 – Secțiunile 4.1, 4.2

Cuprins

- Problematica IPC
- Condiții de cursă; sincronizare
- Regiuni critice
- Semafoare; mutexuri; bariere
- Problema producător-consumator
- Problema cititori-scriitori
- Deadlock-uri

Comunicarea între procese (IPC)

- Colaborare
  - schimb de informație
  - partajarea informației
- Concurență/competiție
  - acces exclusiv (regiuni critice)
    - rezultate predictibile
    - accesul concurent poate produce inconsistențe
    - serializarea accesului
- Coordonare
  - ordonarea acțiunilor unui proces în funcție de acțiunile altui proces
  - sincronizare

Acces exclusiv și sincronizare

- Un proces (P1) folosește cu resursa R
- Un alt proces (P2) solicită resursa R prelucrată de P1
- P2 trebuie să aștepte (**acces exclusiv**) eliberarea resursei R

- P1 folosește R1 și produce R2
- P2 solicită R2
- P2 trebuie să aștepte ca P1 să producă R2 (**sincronizare**)

- Mecanisme de tip eveniment
  - **signal/notify** – notifică producerea evenimentului/acțiunii
  - **wait** – așteaptă producerea evenimentului pentru sincronizare

Acces concurent

- Resursă comună – fișier, zonă de memorie

- Situație

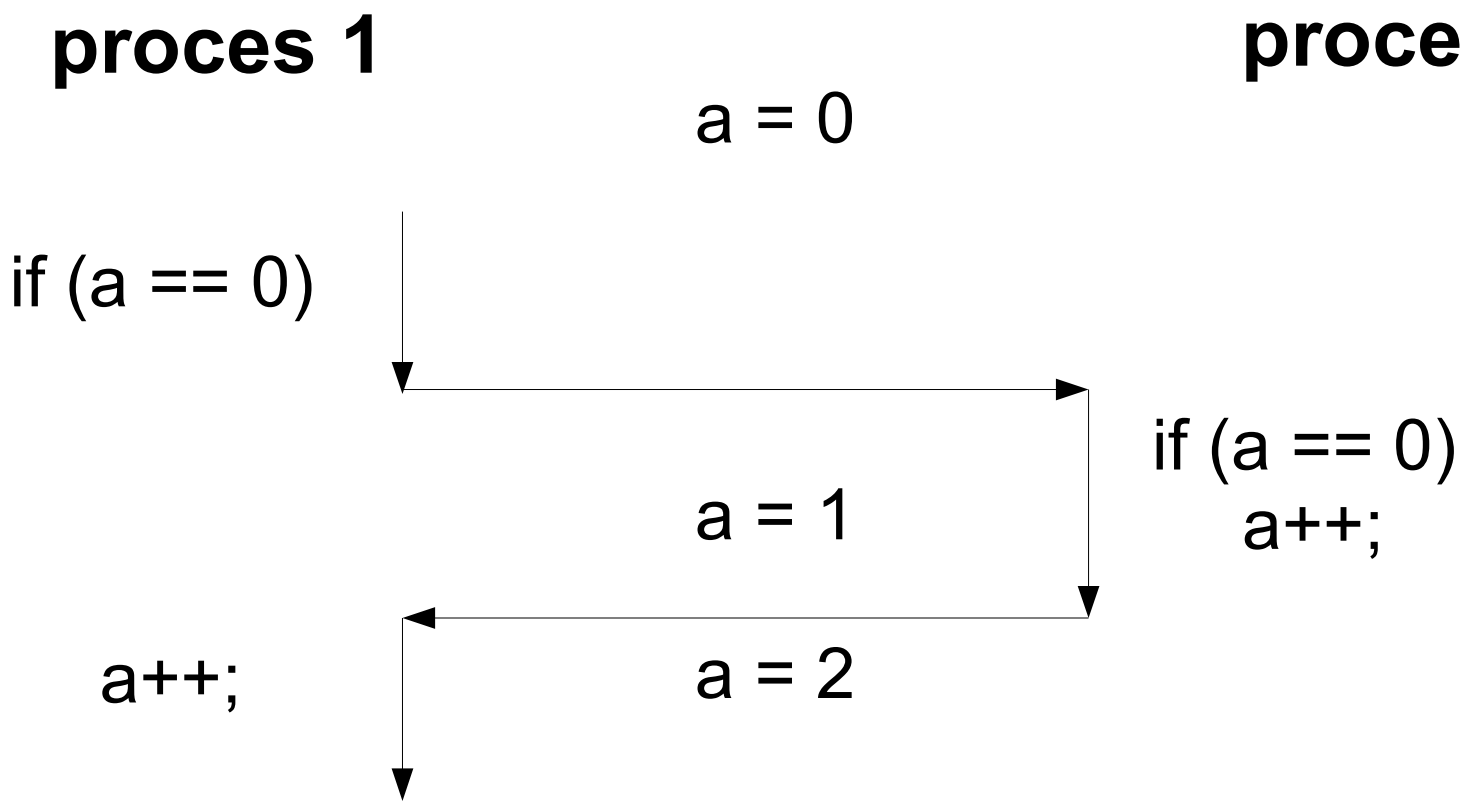
```
a = 0 /* initializare */
```

  - două instanțe de execuție rulează

```
if (a == 0)
   a++;
```

  - Ce valoare va avea a după execuție?

Condiții de cursă (race conditions)



Operații atomice

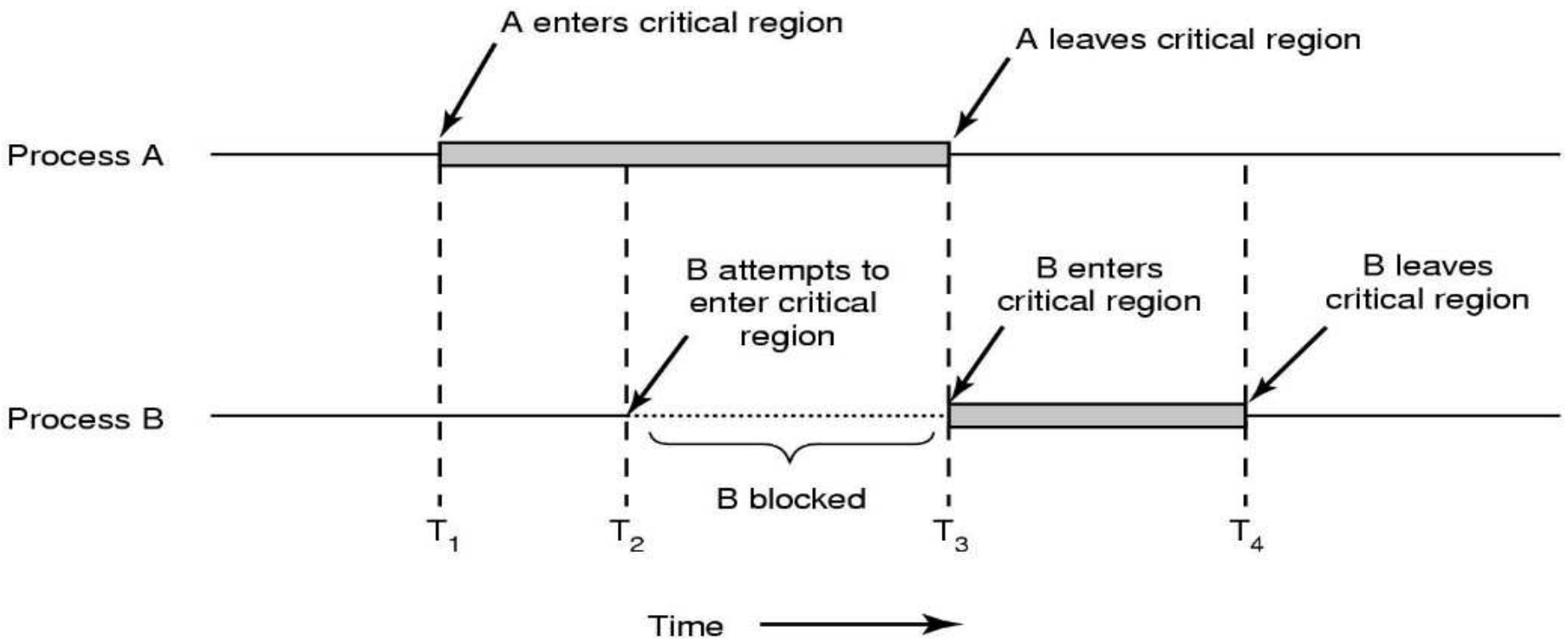
- Operații care se execută într-un singur ciclu de instrucțiune
- În câte cicluri se execută a++?
- Secvență posibilă:

thread1	thread2	a	reg1	reg2
`load a, reg1`		0	0	0
`inc reg1`		0	1	0
	`load a, reg2`	0	1	1
	`inc reg2`	0	1	1
	`store reg2, a`	1	1	1
`store reg1, a`		1	1	1

Regiuni critice

- Critical sections
- Părți din program care accesează resurse partajate
- Pot fi executate în paralel de mai multe instanțe de execuție
- Exemple
  - a++
  - parcurgerea/adăugarea/ștergerea unor elemente dintr-o listă
- Se pot genera condiții de cursă
  - soluție: protejare prin excludere mutuală
    - o singură instanță de execuție are acces la regiunea critică
    - serializarea accesului

Regiuni critice (2)



Excludere mutuală

- Implementare busy-waiting
  - procesul care așteaptă să intre în regiunea critică folosește procesorul
  - testare în buclă a valorii unei variabile
- Implementare blocantă
  - procesul care așteaptă să intre în regiunea critică este trecut în starea BLOCKED
  - când regiunea critică este liberă, procesul este trecut în starea READY/RUNNING

Dezactivarea întreruperilor

- Asigură protecție doar pe sisteme uniprocesor
  - pe un sistem uniprocesor o schimbare de context este inițiată numai prin intermediul unei întreruperi
  - dezactivare întreruperi = nu mai există schimbări de context
- Avantaje/dezavantaje
  - simplitate
  - relativ lentă
  - funcționează doar pe sisteme uniprocesor
  - poate fi folosită doar din cod kernel

Kernel preemptiv/non-preemptiv

- Codul kernel este cod executat de toate procesele
  - structurile de date ale kernel-ului sunt susceptibile la condiții de cursă
- Kernel preemptiv
  - un proces poate fi preemptat în timp ce rulează cod kernel
  - mai complex de implementat
  - timp bun de răspuns
- Kernel non-preemptiv
  - procesul care rulează cod kernel nu poate fi preemptat decât la ieșirea din kernel
  - mai simplu de implementat
  - latență mai mare
  - pot apărea probleme de sincronizare pe sisteme SMP

Lock-uri

```
void LOCK(void)
{
   while (lock > 0)
      continue;
   lock++;
}
```

```
void UNLOCK(void)
{
   lock--;
}
```

- La intrarea în regiunea critică se apelează LOCK
- La ieșirea din regiunea critică se apelează UNLOCK
- Este implementarea corectă?

Alternare strictă

```
while (TRUE) {
   while (turn != 0)
      continue;
   critical_region();
   turn = 1;
   noncritical_region();
}
```

```
while (TRUE) {
   while (turn != 1)
      continue;
   critical_region();
   turn = 0;
   noncritical_region();
}
```

- Procesul mai lent îl ține pe loc pe celălalt

Algoritmul lui Peterson

```
int turn;
int interested[2];

void enter_region(int process)
{
    int other = 1 – process;

    interested[process] = 1;
    turn = process;
    while (turn == process && interested[other] == 1)
        continue;
}

void leave_region(int process)
{
    interested[process] = FALSE;
}
```

Hardware de sincronizare

- TestAndSet/TSL (Test and Set Lock)
- TSL RX, LOCK
  - se citește variabila LOCK în registrul RX
  - se pune valoarea TRUE (activat) în LOCK
  - operația se execută atomic
- Echivalent cu

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

Hardware de sincronizare (2)

- Exemplu de utilizare TestAndSet

```
do {
    while (TestAndSetLock(&lock))
       Continue;
    /* regiune critica */
    lock = FALSE;
    /* regiune non-critica */
} while (TRUE);
```

Spinlock-uri

```
enter_region:
   TSL RX, LOCK
   CMP RX, 0
   JNE enter_region
```

```
exit_region:
   MOV LOCK, 0
```

- Necesită suport hardware
- Pe sistemele multiprocesor se folosesc în combinație cu acces exclusiv la magistrală
- Soluție independentă de numărul de procesoare

Excludere mutuală prin blocare

- De ce nu busy-waiting?
  - irosire timp procesor
- Când este util busy-waiting?
  - când timpul de așteptare este mic
- Excludere mutuală prin blocare
  - **`sleep()`** -> trece procesul în starea BLOCKED
  - **`wakeup()`** -> trece procesul în starea READY
  - necesită suportul scheduler-ului

Problema producător-consumator

- Două tipuri de procese
  - procese producător
  - procese consumator
- Un buffer comun de elemente
- Un producător nu poate produce dacă bufferul este plin
- Un consumator nu poate consuma dacă bufferul este gol

Problema producător-consumator (sleep & wakeup)

```
void producer(void)
{
    int item;
    while (TRUE) {
        item = produce();
        if (count == N)
            Sleep();
        insert_item(item);
        count++;
        if (count == 1)
            wake_up(consumer);
    }
}
```

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

condiție de cursă:
se pierde wake-up-ul

Semafoare

- Rezolvarea problemelor cu sleep și wakeup
- Dijkstra 1965

- Semaforul are asociat un contor
- Operații pe semafor - atomice
  - P (proberen = to test) (down/get/acquire)
    - contorul este decrementat dacă este strict pozitiv
    - altfel, procesul este trecut în starea BLOCKED
  - V (verhogen = to increment) (up/post/release)
    - contorul este incrementat
    - dacă există procese blocate la semafor sunt trezite

Problema producător-consumator (semafoare)

```
semaphore to_full = N;
semaphore to_empty = 0;
```

```
void consumer(void)
{
    int item;
    while(TRUE) {
        P(to_empty);
        item = remove_item();
        V(to_full);
        consume_item();
    }
}
```

```
void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        P(to_full);
        insert_item(item);
        V(to_empty);
    }
}
```

Mutex-uri

- Semafoare binare
- Două stări:
  - locked
  - unlocked
- Asemănător unui spinlock fără busy-waiting

```
P(mutex);
/* regiune critica */
V(mutex)
```

P-C (semafoare și mutex-uri)

```
semaphore to_full = N;
semaphore to_empty = 0;
semaphore mutex = 1;
```

```
void producer(void)
{
    int item;
    while(TRUE) {
        P(to_empty);
        P(mutex);
        item = remove_item();
        V(mutex);
        V(to_full);
        consume_item();
    }
}
```

```
void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        P(to_full);
        P(mutex);
        insert_item(item);
        V(mutex);
        V(to_empty);
    }
}
```

Asigurarea excluderii mutuale la lucrul cu bufferul

Implementare posibilă mutex

```
mutex_lock:
    TSL RX, MUTEX
    CMP RX, 0
    JZE ok
    CALL sched_yield
    JMP mutex_lock
ok:
```

```
mutex_unlock:
    MOV MUTEX, 0
    ...
```

- Similară cu implementarea spinlock-urilor (non-busy waiting)
- Simplă și eficientă (nu necesită trap în kernel dacă regiunea critică este liberă)

Implementare semafor

```
typedef struct {
    int value;
    struct process *list;
}
void down(semaphore *s)
{
    s->value--;
    if (s->value < 0) {
        add(s->list, current_process);
        sleep();
    }
}
void up(semaphore *s)
{
    if (s->value < 0) {
        process = remove_process(s->list);
        wakeup(process);
    }
    s->value++;
}
```

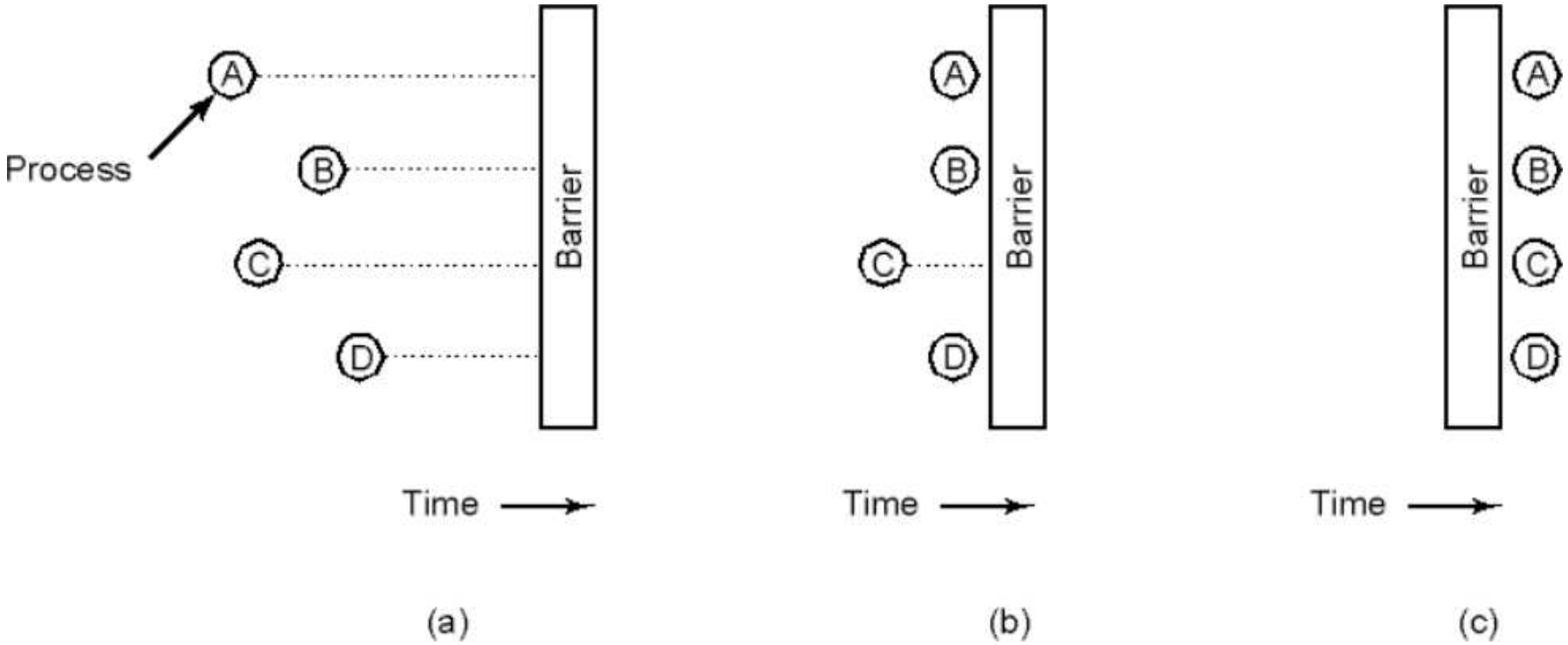
Probleme de sincronizare

- Deadlock
  - două procese așteaptă la un semafor deținut de celălalt proces
  - P0: down(&sem_a); down(&sem_b); ...
  - P1: down(&sem_b); down(&sem_a); ...
  - P0 este planificat înainte de obținerea lui sem_b
- Starvation
  - adăugarea/eliminarea proceselor din lista de procese a unui semafor se face în ordinea LIFO
  - așteptare nedefinită
  - priorități statice

Bariere

- Sincronizarea unui grup de procese
- Utile pentru rezolvarea de probleme inginerești
  - o matrice conține valori inițiale și mai multe prelucrări succesive care trebuie aplicate
  - pentru ca un proces să înceapă iterația n, trebuie ca toate procesele să încheie iterația n+1
- Punctul de întâlnire a proceselor/thread-urilor se numește *rendezvous*

Bariere (2)


Process
A
B
C
D
Barrier
Time
(a)
A
B
C
D
Barrier
Time
(b)
Barrier
A
B
C
D
Time
(c)

Barieră (3)

```
#define N_PROCS   10

size_t count = 0;
semaphore mutex = 1;
semaphore barr_sem = 0;

void barrier(void)
{
    down(&mutex);
    count++;
    if (count == N_PROCS)
        up(&barr_sem, N_PROCS);
    up(&mutex);

    down(&barr_sem);
}
```

```
/* UNIX semaphore Z-operation */

#define N_PROCS   10

semaphore barr_sem = N_PROCS;

void barrier(void)
{
    down(&barr_sem);
    wait_for_zero(&barr_sem);
}
```

Barieră reentrantă

```
#define N_PROCS   10

size_t count = 0;
semaphore mutex = 1;
semaphore barr_sem = 0;

void rbarrier(void)
{
    down(&mutex);
    count++;
    if (count == N_PROCS) {
        up(&barr_sem, N_PROCS);
        count = 0;
    }
    up(&mutex);

    down(&barr_sem);
}
```

- Care este problema?
- Procesul care dă drumul barierei este preemptat după up(&mutex)
- Primul proces din noua rundă trece de barieră

Barieră reentrantă (2)

```
#define N_PROCS   10

size_t count = 0;
semaphore mutex = 1;
semaphore barr_sem_p1 = 0;
semaphore barr_sem_p2 = 0;

void rbarrier_phase1(void)
{
    down(&mutex);
    count++;
    if (count == N_PROCS)
        up(&barr_sem_p1, N_PROCS);
    up(&mutex);

    down(&barr_sem_p1);
}
```

```
void rbarrier_phase2(void)
{
    down(&mutex);
    count--;
    if (count == 0)
        up(&barr_sem_p2, N_PROCS);
    up(&mutex);

    down(&barr_sem_p2);
}

/* usage skeleton */
for(...) {
    /* rendezvous */
    rbarrier_phase1();
    /* critical section */
    rbarrier_phase2();
}
```

Problema cititori-scriitori

- Situația în care diverse instanțe de execuție solicită acces la o resursă (fișier/memorie/bază de date)
- Acces de citire sau de scriere
- Constrângerile de sincronizare
  - oricâți cititori pot fi în regiunea critică la un moment dat
  - scriitorii trebuie să aibă acces exclusiv la regiunea critică
- Utilă în aplicațiile cu mulți cititori

Problema cititori-scriitori (2)

```
do {
    down(&excl);
    ...
    /* do writing */
    ...
    up(&excl);
} while (TRUE);
```

```
do {
    down(&mutex);
    read_count++;
    if (read_count == 1)
        down(&excl);
    up(&mutex);
    ...
    /* do reading */
    ...
    down(&mutex)
    read_count--;
    if (read_count == 0)
        up(&excl);
    up(&mutex);
} while (TRUE);
```

Problema cititori-scriitori (3)

- Problemă la soluția precedentă
  - posibil starvation de scriitori
  - dacă există un flux constant de cititori, scriitorii nu mai intră în regiunea critică
- Încercați
  - implementarea unei soluții care să evite starvation
  - implementarea unei soluții cu prioritate pe scriitori
  - încercați, în primă fază, să nu folosiți “Little Book of Semaphores”

P-C (transfer de mesaje)

```
void consumer(void)
{
    int item, i;
    message m;

    while(TRUE) {
        receive(producer, &m);
        item = extract_item(m);
        consume_item();
    }
}
```

```
void producer(void)
{
    int item, i;
    message m;

    while (TRUE) {
        item = producer_item();
        build_msg(&m, item);
        send(consumer, &m);
    }
}
```

Deadlock - modelare

- Număr finit de resurse (memorie, I/O, CPU)
  - pot exista instanțe de resurse (2 CPU, 5 imprimante etc.)
- Set de procese care solicită concurent accesul la resurse
  - cerere (request) de folosire a resursei
    - poate fi furnizată imediat sau trebuie să aștepte eliberarea resursei de alt proces
    - malloc/open
  - folosire (use)
  - eliberare (release)
    - free/close

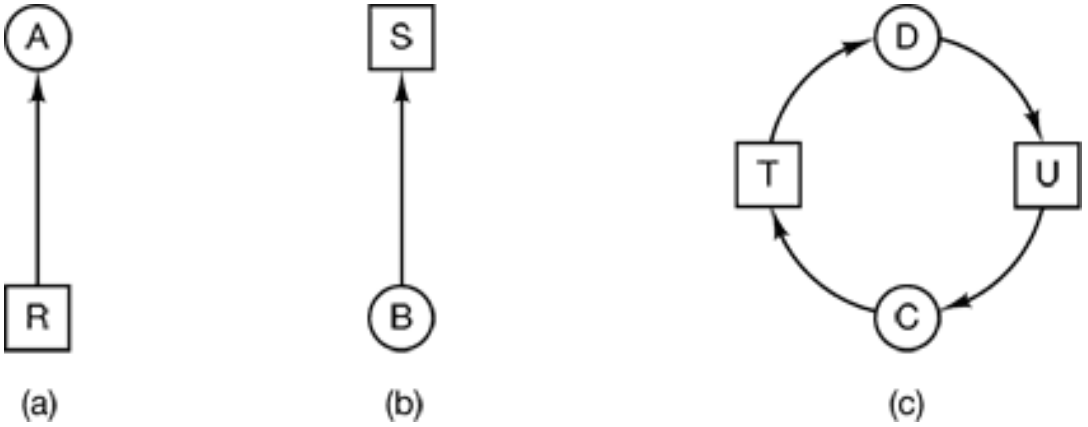
Deadlock – condiții necesare

- Excludere mutuală
  - o resursă este deținută în mod non-partajat
- Deține și așteaptă (hold & wait)
  - un proces deține o resursă și așteaptă obținerea altor resurse deținute de alte procese
- Fără preemptare
  - resursele nu pot fi preemptate – un proces eliberează o resursă doar voluntar
- Așteptare circulară (circular wait)
  - există o mulțime {P0, P1, ..., Pn} de procese
    - P0 așteaptă o resursă deținută de P1
    - P1 așteaptă o resursă deținută de P2
    - ... Pn așteaptă o resursă deținută de P0

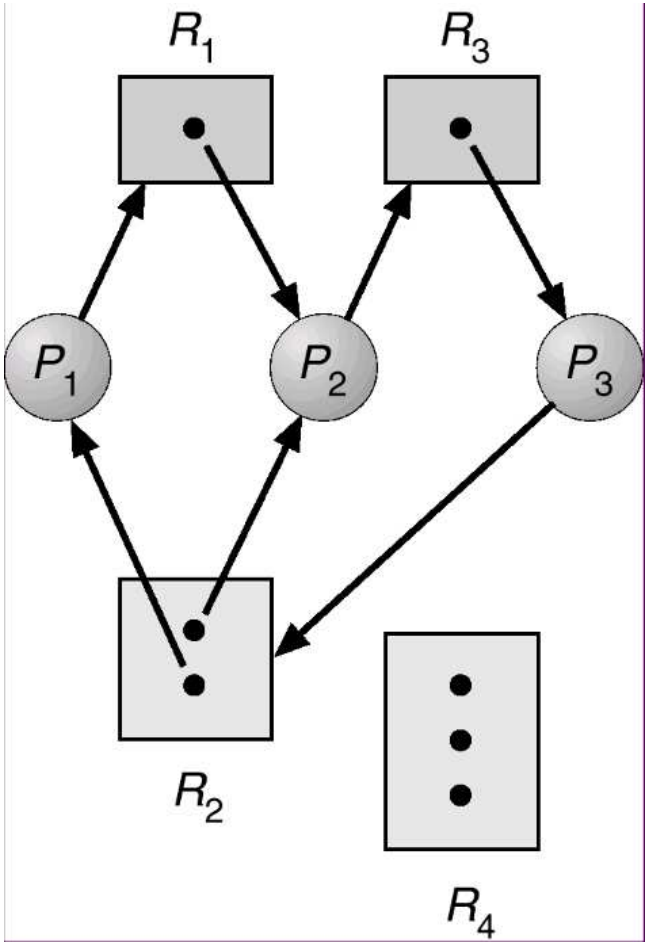
Graful de alocare a resurselor

- Nodurile reprezintă
  - resurse – pătrate
  - procese – cercuri
- Arcele reprezintă
  - resursă deținută – arc de la resursă la proces
  - resursă dorită – arc de la proces la resursă
- Se poate demonstra că
  - dacă nu există ciclu în graf, atunci nu există deadlock
  - dacă există ciclu în graf, este posibil să existe deadlock

Graful de alocare a resurselor (2)



(a) procesul A deține resursa R
(b) procesul B solicită resursa S
(c) deadlock



Detectare deadlock

- Se analizează graful de alocare a resurselor
- Se descoperă ciclurile din graf
- Este posibil să existe ciclu dar nu deadlock
- Există algoritmi formali care pot detecta deadlock-urile din sistem
- Algoritmul struțului (UNIX, Windows)
  - “nu-mi pasă” (laissez-faire)

Evitare deadlock

- Se cunoaște ordinea în care un proces solicită resursele
- Pe baza acestor cunoștințe se poate planifica accesul la resurse
- Secvență sigură
  - o secvență de proces <P1, P2, ..., Pn>
  - solicitările lui Pi pot fi satisfăcute cu resursele actuale libere sau cele deținute de Pj (j < i)
- Stare sigură (safe state) a sistemului
  - există o secvență sigură pentru procesele sistemului
- Stare nesigură - deadlock

Prevenire deadlock

- Una din cele 4 condiții necesare pentru existența unui deadlock nu este satisfăcută

- Excludere mutuală
  - greu de eliminat
  - o bună parte din resurse sunt intrinsec non-partajabile

- Hold and wait
  - se cer toate resursele de la început

Prevenire deadlock (2)

- No preemption
  - dacă un proces solicită resurse ocupate, eliberează resursele proprii
- Așteptare circulară
  - ordonare a resurselor (R1 < R2 < ... < Rn)
  - dacă un proces solicită Ri trebuie să solicite și Rj (j < i)
  - dacă un proces eliberează Rj trebuie să elibereze Ri (j < i)

Recuperare din deadlock

- Terminarea procesului
  - toate procesele care “participă” la deadlock
  - câte un proces până la dispariția deadlock-ului
- Preemptarea resurselor
  - preemptare resurse de la anumite procese
  - se oferă resursele altor procese

Cuvinte cheie

- condiție de cursă
- acces exclusiv
- sincronizare
- regiune critică
- TSL
- spinlock
- semafor
- mutex
- barieră
- barieră reentrantă
- producător-consumator
- cititori-scriitori
- deadlock
- graf de alocare a resurselor
- prevenire/evitare deadlock

Exercițiu 1

- Descrieți o situație în care un proces poate fi planificat într-un kernel non-preemptiv în momentul în care execută cod kernel.

- De ce folosirea de priorități statice poate conduce la starvation?

Exercițiu 2

- Care din următoarele operații apelează planificatorul de procese/thread-uri?
  - citirea dintr-un fișier
  - deschiderea unui fișier
  - operația P pe un semafor
  - operația V pe un semafor
  -

Întrebări

?