



di.unito.it

DIPARTIMENTO
DI INFORMATICA

**laboratorio di
sistemi operativi**

semafori

Marco Botta

Materiale preparato da Daniele Radicioni

argomenti del laboratorio UNIX

- introduzione a UNIX;
- integrazione C: operatori bitwise, precedenze, preprocessore, pacchettizzazione del codice, compilazione condizionale e utility make;
- controllo dei processi;
- segnali;
- pipe e fifo;
- code di messaggi;
- semafori;
- memoria condivisa;
- introduzione alla programmazione bash.



credits

- il materiale di queste lezioni è tratto:
 - dai lucidi del Prof. Gunetti degli anni scorsi;
 - Michael Kerrisk, *The Linux Programming interface - a Linux and UNIX® System Programming Handbook*, No Starch Press, San Francisco, CA, 2010;
 - W. Richard Stevens (Author), Stephen A. Rago, *Advanced Programming in the UNIX® Environment (2nd Edition)*, Addison-Wesley, 2005;



introduzione: i semafori

a che servono

- diversamente dagli altri strumenti per IPC visti finora, i semafori **non servono a trasferire dati fra processi.**
- sono utilizzati per permettere ai processi di **sincronizzare le proprie azioni**: per esempio permettono di sincronizzare l'accesso a un blocco di memoria condivisa, per impedire a un processo di accedere alla memoria condivisa mentre un altro processo la sta aggiornando



cos'è un semaforo

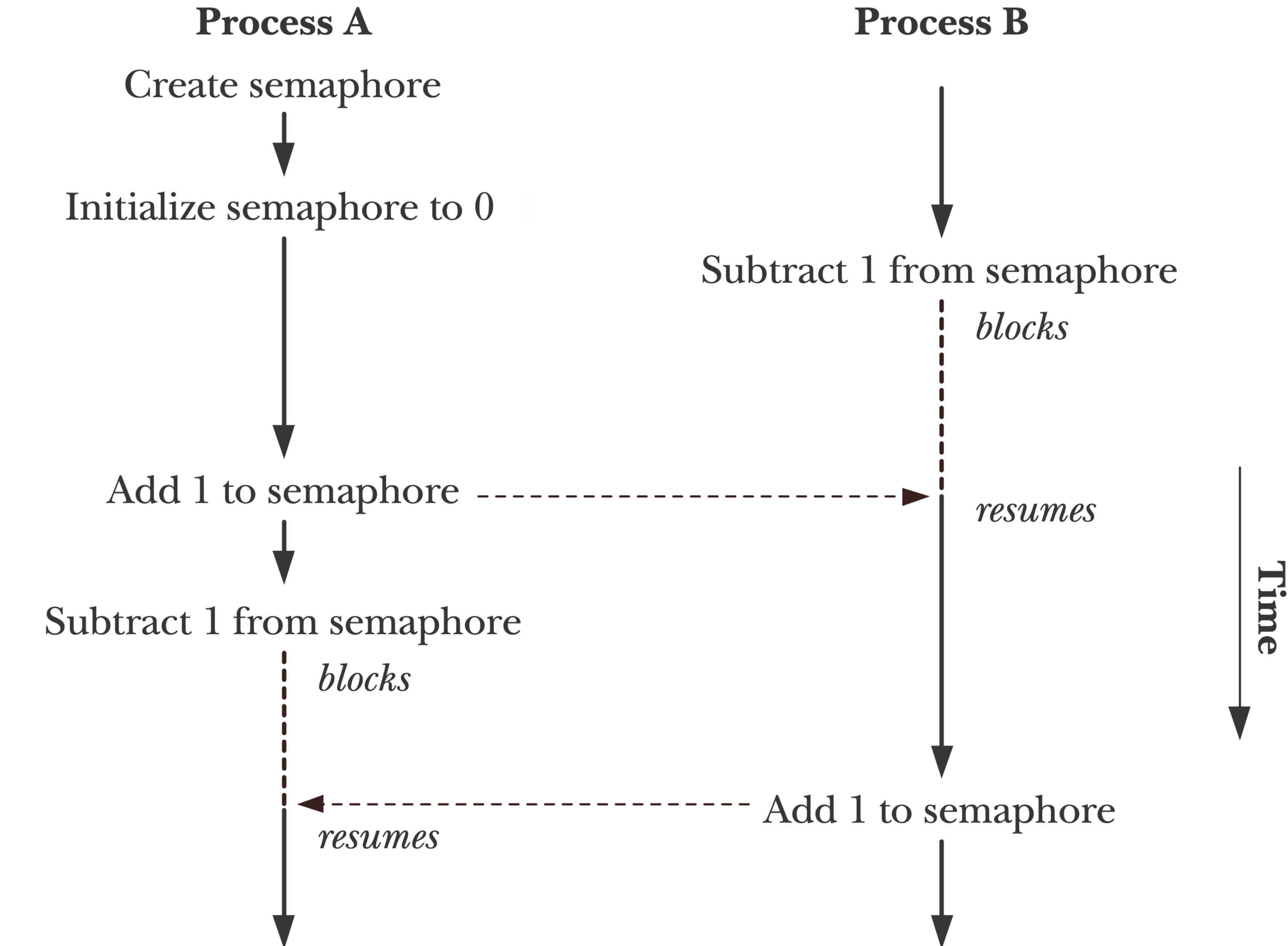
- un semaforo è un **intero mantenuto dal kernel il cui valore è sempre ≥ 0 .**
 - è possibile effettuare varie operazioni su un semaforo, fra cui:
 1. impostare il semaforo ad un certo valore;
 2. aggiungere un numero al valore corrente del semaforo;
 3. sottrarre un numero dal valore corrente del semaforo; e
 4. attendere che il valore del semaforo sia 0.



cos'è un semaforo

- le ultime due operazioni (sottrarre un numero e attendere lo 0) possono causare il blocco del chiamante.
 - nel decrementare il valore del semaforo il kernel blocca qualsiasi tentativo di ridurlo sotto 0.
 - analogamente, l'attesa dello 0 blocca il chiamante se il semaforo non è attualmente a 0.
- in entrambi i casi, il chiamante resta bloccato finché qualche altro processo modifica il semaforo a un valore che consente all'operazione di procedere; a questo punto il kernel risveglia il processo bloccato.





utilizzo dei semafori

- i passi per utilizzare i semafori sono:
 - **creazione o apertura** di un set di semafori utilizzando la `semget()`.
 - **inizializzazione** dei semafori presenti nel set, usando l'operazione `SETVAL` o `SETALL` della `semctl()`.
 - **esecuzione delle operazioni** sui valori del semaforo, utilizzando `semop()`.
- i processi che utilizzano il semaforo tipicamente usano tali operazioni per indicare l'acquisizione e il rilascio di una risorsa condivisa.
- quando tutti i processi hanno terminato di usare il set di semafori, **rimozione del set** per mezzo dell'operazione `IPC_RMID` della `semctl()`.



set di semafori

- i semafori di System V sono resi complessi dal fatto di essere allocati in gruppi detti **set di semafori**.
 - il numero di semafori in un set è specificato al momento della creazione del set, per mezzo della system call `semget()`.
 - molto spesso si utilizza un solo semaforo alla volta, ma la system call `semop()` consente di eseguire atomicamente un gruppo di operazioni su vari semafori in uno stesso set.



```

00 int main(int argc, char *argv[]) {
01     int semid;
02     if (argc == 2) { // ----- creo e inizializzo un sem -----
03         union semun arg;
04         $ ./handle_sem 0
05         creato e inizializzato semaforo con ID = 371130369
06         $ ./handle_sem 371130369 -2 &
07         [1] 71130
08         about to semop at 2024-11-02_04:59:14.
09         $ ./handle_sem 371130369 +3
10         71136: about to semop at 2024-11-02_04:59:47.
11         71136: semop completed at 2024-11-02_04:59:47.
12         71130: semop completed at 2024-11-02_04:59:47.
13         [1]+ Done          ./handle_sem 371130369 -2
14         $
15
16         // operazioni speciali
17         $ printf("%ld: about to semop at %s\n",
18         [1]+(long) getpid(), get_curr_time()); 371130369 -2
19         71130: semop(semid, &sem, 1) == -1
20         addErrExit("semop");
21         printf("%ld: semop completed at %s\n",
22             (long) getpid(), get_curr_time());
23     }
24
25     exit(EXIT_SUCCESS);

```



```

00 int main(int argc, char *argv[]) {
01     int semid;
02     if (argc == 2) { // ----- creo e inizializzo un sem -----
03         union semun arg;
04         semid = semget(IPC_PRIVATE, 1, S_IRUSR | S_IWUSR);
05         if (semid == -1) ddErrExit("semid");
06         arg.val = atoi(argv[1]);
07         if (semctl(semid, /* semnum= */ 0, SETVAL, arg) == -1)
08             ddErrExit("semctl");
09         printf("creato e inizializzato semaforo con ID = %d\n", semid);
10     } else { // ----- eseguo un'operazione sul primo semaforo -----
11         struct sembuf sop; // la struttura che definisce l'operazione
12         semid = atoi(argv[1]);
13         sop.sem_num = 0; // specifica il primo semaforo nel set
14         sop.sem_op = atoi(argv[2]);
15             // aggiungo, sottraggo o attendo 0
16         sop.sem_flg = 0; // per ora NON settiamo flag per effettuare
17             // operazioni speciali
18         printf("%ld: about to semop at %s\n",
19             (long) getpid(), get_curr_time());
20         if (semop(semid, &sop, 1) == -1)
21             ddErrExit("semop");
22         printf("%ld: semop completed at %s\n",
23             (long) getpid(), get_curr_time());
24     }
25     exit(EXIT_SUCCESS); }
```



Creating or Opening a Semaphore Set

```
#include <sys/types.h> /* For portability */  
#include <sys/sem.h>  
  
int semget(key_t key, int nsems, int semflg);
```

Returns semaphore set identifier on success,
or -1 on error

- l'argomento **key** è una chiave generata specificando un numero casuale, o usando il valore **IPC_PRIVATE** o una chiave restituita da **ftok()**.
 - se stiamo usando **semget()** per creare un nuovo set di semafori, allora **nsems** specifica il numero di semafori in quell'insieme, e deve essere maggiore di 0. se invece stiamo usando la **semget()** per ottenere l'identificatore di un set esistente, **nsems** deve essere minore o uguale alla dimensione del set (o si incorrerà nell'errore **EINVAL**).
- NB: non è possibile modificare il numero di semafori presenti in un dato set.



```
#include <sys/types.h> /* For portability */
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Returns semaphore set identifier on success,
or -1 on error

- **l'argomento semflg è una maschera di bit che specifica i permessi da assegnare a un nuovo set di semafori o da verificare su un set esistente.**

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute



```
#include <sys/types.h> /* For portability */  
#include <sys/sem.h>  
  
int semget(key_t key, int nsems, int semflg);
```

Returns semaphore set identifier on success,
or -1 on error

- zero o più dei seguenti flag possono essere messi in OR (|) nel semflg per controllare l'operazione **semget()**:
 - **IPC_CREAT**. se non esiste un set di semafori con la chiave specificata, crea un nuovo set.
 - **IPC_EXCL**. se è stato specificato anche **IPC_CREAT**, e un set di semafori con la chiave specificata esiste già, restituisce un fallimento, con errore EEXIST.



Semaphore Control Operations

```
#include <sys/types.h> /* For portability */  
#include <sys/sem.h>  
  
int semctl(int semid, int semnum,  
            int cmd, ... /* union semun arg */);
```

Returns nonnegative integer on success;
returns -1 on error

- l'argomento **semid** è l'**identificatore del set di semafori** sul quale l'operazione viene eseguita.
- per le operazioni su un singolo semaforo l'argomento **semnum** identifica un **semaforo all'interno del set**. per le altre operazioni questo argomento è ignorato, e possiamo lasciarlo a 0.
- l'argomento **cmd** specifica l'**operazione**.



```
#include <sys/types.h> /* For portability */
#include <sys/sem.h>

int semctl(int semid, int semnum,
            int cmd, ... /* union semun arg */);
```

Returns nonnegative integer on success (see text);
returns -1 on error

- alcune operazioni richiedono un quarto argomento di nome **arg**.
- si tratta di una **union** che deve essere esplicitamente definita nei nostri programmi (a meno che non sia già presente in `<sys/types.h>`, come in Mac OS X).

```
union semun {
    // value for SETVAL
    int val;
    // buffer for IPC_STAT, IPC_SET
    struct semid_ds* buf;
    // array for GETALL, SETALL
    unsigned short* array;
    // Linux specific part
#ifndef __linux__
    // buffer for IPC_INFO
    struct seminfo* _buf;
#endif
};
```



Generic control operations

```
int semctl(int semid, int semnum,  
           int cmd, ... /* union semun arg */);
```

- le seguenti operazioni (IPC_RMID, IPC_STAT, IPC_SET) sono le stesse applicabili agli altri tipi di oggetti IPC di System V.
 - **IPC_RMID**. rimuove **immediatamente** il set di semafori e l'associata struttura `semid_ds`. qualsiasi processo bloccato in chiamate `semop()` in attesa su semafori è **immediatamente svegliato**, e `semop()` riporta l'errore `EIDRM`.
 - l'argomento `arg` non è richiesto.



Generic control operations

```
int semctl(int semid, int semnum,  
           int cmd, ... /* union semun arg */);
```

- **IPC_STAT**. copia la struttura `semid_ds` associata con il set di semafori nel buffer puntato da `arg.buf`.
- **IPC_SET**. aggiorna i membri della struttura `semid_ds` associata al set di semafori utilizzando i valori nel buffer puntato da `arg.buf`.



Retrieving and initializing semaphore values

```
int semctl(int semid, int semnum,  
            int cmd, ... /* union semun arg */);
```

- le seguenti operazioni prelevano o inizializzano il valore (o i valori) di un singolo semaforo o di tutti i semafori del set.
 - la copia del valore di un semaforo richiede permessi in lettura sul semaforo, mentre l'inizializzazione del valore richiede permessi in scrittura.
 - **GETVAL**. come risultato della chiamata, **semctl()** restituisce il valore del semaforo (numero) **semnum** nel set di semafori specificato da **semid**.
l'argomento **arg** non è richiesto.
 - **SETVAL**. il valore del semaforo **semnum** nel set riferito da **semid** è **inizializzato al valore **arg.val****.



```
int semctl(int semid, int semnum,  
           int cmd, ... /* union semun arg */);
```

```
union semun {  
    int val;  
    struct semid_ds* buf;  
    unsigned short* array;  
#if defined(_linux_)  
    struct seminfo* _buf;  
#endif  
};
```

- le seguenti operazioni prelevano o inizializzano il valore (i valori) di un singolo semaforo o di tutti i semafori nel set.
 - **GETALL**. preleva i valori di tutti i semafori nel set riferito da semid, copiandoli nell'array puntato da **arg.array**. il programmatore deve garantire che l'array sia abbastanza capiente. semnum è ignorato.
 - **SETALL**. inizializza tutti i semafori del set riferito da semid, usando i valori forniti nell'array puntato da **arg.array**. semnum è ignorato.



Retrieving per-semaphore information

- le seguenti operazioni restituiscono (attraverso il valore restituito dalla funzione) **informazioni sul semaforo** numero **semnum** del set riferito da **semid**.
- per tutte queste operazioni, è richiesto il permesso in lettura sul set di semafori, e l'argomento **arg** non è richiesto.
- **GETPID**. restituisce il PID dell'ultimo processo che ha eseguito una **semop()** su questo semaforo; questo è riferito come il valore **sempid**. se nessun processo ha ancora eseguito una **semop()** su questo semaforo, restituisce 0.
- **GETNCNT**. restituisce il numero di processi attualmente **in attesa di un incremento del valore del semaforo**; questo è riferito come il valore **semncnt**.
- **GETZCNT**. restituisce il numero di processi attualmente **in attesa che il valore del semaforo divenga 0**; questo è riferito come valore **semzcnt**.



```

struct semid_ds {
    /* Ownership and permissions */
    struct ipc_perm sem_perm;
    /* Last semop time */
    time_t          sem_otime;
    /* Last change time */
    time_t          sem_ctime;
    /* No. of semaphores in set */
    unsigned short  sem_nsems;
} ;

```

```

union semun {
    int val;
    struct semid_ds* buf;
    unsigned short* array;
#endif defined(__linux__)
    struct seminfo* __buf;
#endif
};

```

- ogni set di semafori ha associata una struttura `semid_ds` data;
- i membri della struttura `semid_ds` sono implicitamente aggiornati da varie system call sul semaforo, e alcuni membri della struttura `sem_perm` possono essere aggiornati esplicitamente con l'operazione `semctl()` `IPC_SET`.



```
struct semid_ds {  
    struct ipc_perm sem_perm; /* Ownership and permissions */  
    time_t sem_otime; /* Last semop time */  
    time_t sem_ctime; /* Last change time */  
    unsigned short sem_nsems; /* No. of semaphores in set */  
};
```

- **sem_perm**. quando il set di semafori è creato, i membri di questa struttura sono inizializzati. i membri uid, gid, e mode possono essere aggiornati con l'operazione IPC_SET.
- **sem_otime**. questo membro è settato a 0 alla creazione del set di semafori, ed aggiornato all'ora corrente ad ogni semop() che va a buon fine, o quando il valore del semaforo è modificato in seguito a un'operazione SEM_UNDO.
- **sem_ctime**. questo membro è impostato all'ora corrente al momento della creazione del semaforo, e in seguito a ogni operazione IPC_SET, SETALL, o SETVAL.
- **sem_nsems**. membro inizializzato al momento della creazione del set di semafori: contiene il numero di semafori nel set.



esercizio

- programma per monitorare lo stato dei semafori: prende in input l'ID, e stampa quando è stato modificato, l'orario dell'ultima semop(), e poi crea una tabella (utilizzare le opzioni di formattazione per allineare i valori) in cui sono riportati:
 - numero semaforo; valore; SEMPID, SEMNCNT e SEMZCNT.
 - GETNCNT. Return the number of processes currently waiting for the value of this semaphore to increase; this is referred to as the semncnt value.
 - GETZCNT. Return the number of processes currently waiting for the value of this semaphore to become 0; this is referred to as the semzcnt value.



problemi di inizializzazione e race condition

```

00 semid = semget(key, 1, IPC_CREAT | IPC_EXCL | perms);
01 if (semid != -1) { // semaforo creato
02     union semun arg;
03
04     // ----- XXXXXXXX
05
06     arg.val = 0; // inizializzazione
07     if (semctl(semid, 0, SETVAL, arg) == -1)
08         ddErrExit("semctl");
09 } else { // --- non abbiamo creato il sem: esisteva già?
10     if (errno != EEXIST) { // errore inatteso dalla semget()
11         ddErrExit("semget 1");
12     } else { // errore era EEXIST: semaforo già esistente
13         semid = semget(key, 1, perms); // preleviamone l'ID
14         if (semid == -1)
15             ddErrExit("semget 2");
16     }
17 }
18 // esecuzione di un'azione sul semaforo
19 sops[0].sem_op = 1; // aggiunta di 1
20 sops[0].sem_num = 0; // al semaforo 0
21 sops[0].sem_flg = 0; // senza specificare opzioni
22 if (semop(semid, sops, 1) == -1) ddErrExit("semop");

```

race condition

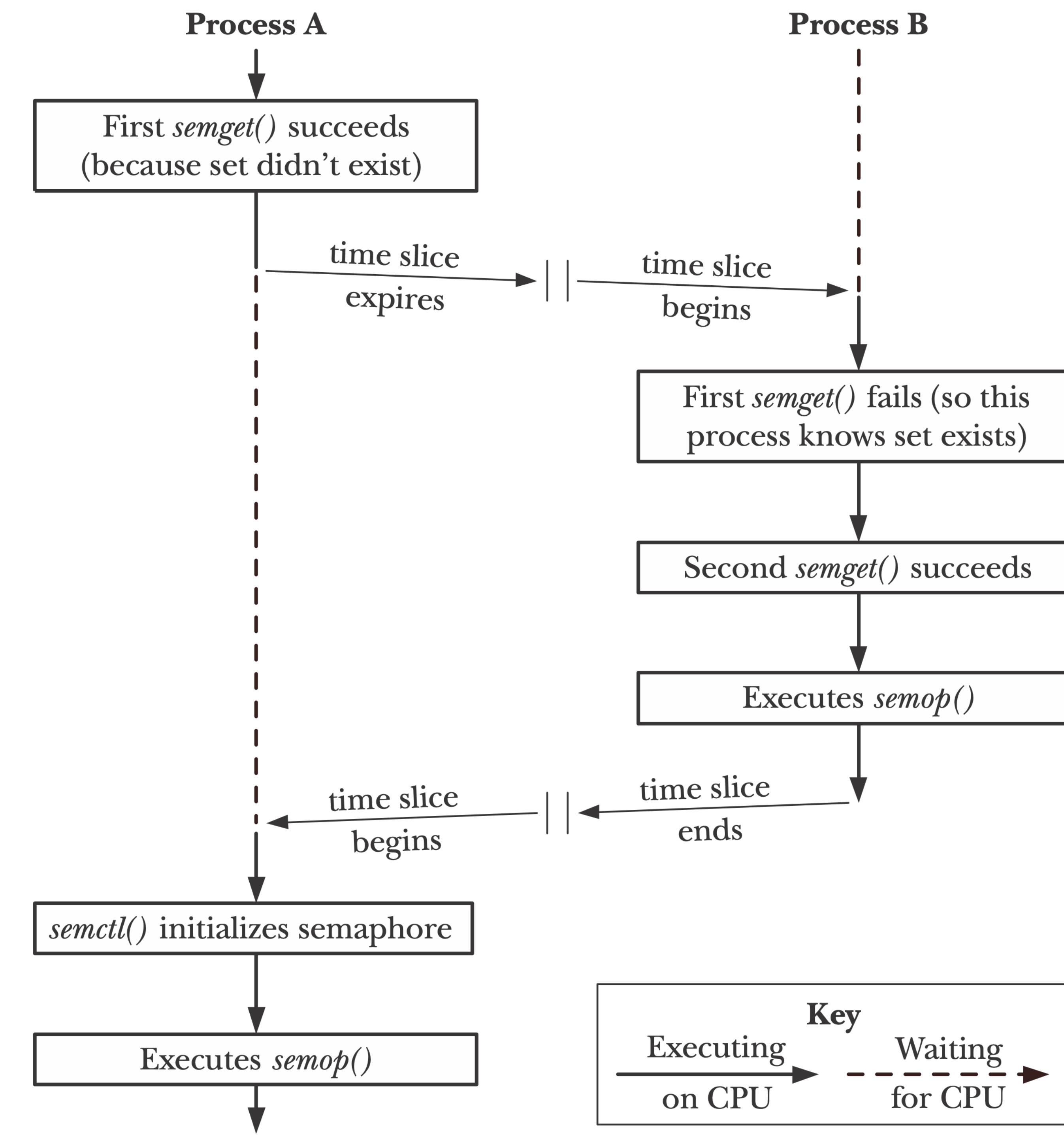
- se due processi eseguono il codice precedente allo stesso tempo, potrebbe verificarsi una sequenza errata se il primo processo viene interrotto al punto marcato nel codice con **XXXX**. questa sequenza pone i seguenti problemi.
 - il processo **B** esegue una **semop()** su un semaforo non inizializzato (cioè, che contiene un valore arbitrario!).
 - la chiamata **semctl()** nel processo **A** sovrascrive le modifiche fatte dal processo **B**.



```

00 semid = semget(key, 1, IPC_CREAT | IPC_EXCL | perms);
01 if (semid != -1) { // semaforo creato
02     union semun arg;
03
04     // ----- XXXXXXXX
05
06     arg.val = 0; // inizializzazione
07     if (semctl(semid, 0, SETVAL, arg) == -1)
08         ddErrExit("semctl");
09 } else { // --- non abbiamo creato il sem: esisteva già?
10     if (errno != EEXIST) { // errore inatteso dalla semget()
11         ddErrExit("semget 1");
12     } else { // semaforo già esistente
13         semid = semget(key, 1, perms); // preleviamone l'ID
14         if (semid == -1)
15             ddErrExit("semget 2");
16     }
17 }
18 // esecuzione di un'azione sul semaforo
19 sops[0].sem_op = 1; // aggiunta di 1
20 sops[0].sem_num = 0; // al semaforo 0
21 sops[0].sem_flg = 0;
22 if (semop(semid, sops, 1) == -1) ddErrExit("semop");

```



soluzione

- la soluzione a questo problema è basata su una caratteristica dell'inizializzazione del membro `sem_otime` della struttura `semid_ds` associata con il set di semafori.
- al momento della creazione del set di semafori, il membro `sem_otime` è inizializzato a 0, ed è modificato solo da una successiva chiamata alla `semop()`.
- possiamo quindi eliminare la race condition inserendo del codice per **forzare il secondo processo** (quello che **NON** crea il semaforo) ad attendere finché il primo processo ha inizializzato il semaforo ed eseguito **la propria `semop()` che aggiorni il membro `sem_otime`.**



```

00 semid = semget(key, 1, IPC_CREAT | IPC_EXCL | perms);

01 if (semid != -1) { // ---- semaforo creato
02     union semun arg;
03     struct sembuf sop;

04     arg.val = 0; // inizializzazione
05     if (semctl(semid, 0, SETVAL, arg) == -1)
06         ddErrExit("semctl 1");

07     /* esecuzione di operazione "no-op". modifico
08      sem_otime in modo che gli altri processi vedano che
09      non è ancora inizializzato */
10     sop.sem_num = 0; // semaforo numero 0
11     sop.sem_op = 0; // attendi per il valore 0
12     sop.sem_flg = 0;
13     if (semop(semid, &sop, 1) == -1)
14         ddErrExit("semop");

15 } else { // ---- non abbiamo creato il semaforo
16     ...
17 }
18
19
20
21

```



```

00 semid = semget(key, 1, IPC_CREAT | IPC_EXCL | perms);

01 if (semid != -1) { // ---- semaforo creato
02     union semun arg;
03     struct sembuf sop;

04     arg.val = 0; // inizializzazione
05     if (semctl(semid, 0, SETVAL, arg) == -1)
06         ddErrExit("semctl 1");

07     /* esecuzione di operazione "no-op". modifico
08      sem_otime in modo che gli altri processi vedano che
09      non è ancora inizializzato */
10     sop.sem_num = 0; // semaforo numero 0
11     sop.sem_op = 0; // attendi per il valore 0
12     sop.sem_flg = 0;
13     if (semop(semid, &sop, 1) == -1)
14         ddErrExit("semop");

15 } else { // ---- non abbiamo creato il semaforo ←
16     ...
17 }
18
19
20
21

```



```

00 if (semid != -1) { // ---- semaforo creato
01 ...
02 } else { // ---- non abbiamo creato il semaforo ←
03     if (errno != EEXIST) ddErrExit("semget 1");
04     // il sem era già stato creato
05     const int MAX_TRIES = 10;
06     int j;
07     union semun arg;
08     struct semid_ds ds;
09     semid = semget(key, 1, perms); // preleviamo l'ID
10     if (semid == -1) ddErrExit("semget 2");
11     // Attesa che un altro processo chiavi semop()
12     arg.buf = &ds;
13     for (j = 0; j < MAX_TRIES; ++j) {
14         if (semctl(semid, 0, IPC_STAT, arg) == -1)
15             ddErrExit("semctl 2");
16         if (ds.sem_otime != 0) // è stata eseguita Semop() ?
17             break; // sì, usciamo dal ciclo
18         sleep(1); // diversamente, attendiamo e ritentiamo
19     }
20     if (ds.sem_otime == 0) // siamo usciti ma sem_otime è 0
21         ddErrExit("Existing semaphore not initialized");
22 }
```



```

00 semid = semget(key, 1, IPC_CREAT | IPC_EXCL | perms);
01
02 if (semid != -1) { // ---- semaforo creato
03     /* esecuzione di operazione "no-op". modifco
04     sem_otime in modo che gli altri processi vedano che
05     non è ancora inizializzato */
06 } else { // ---- non abbiamo creato il semaforo
07     // il sem era già stato creato
08     // Attesa che un altro processo chiavi semop()
09 }
10 // esecuzione di un'azione sul semaforo
11 sops[0].sem_num = 0; /* operazione su semaforo 0... */
12 sops[0].sem_op = (short) atoi(argv[1]);
13 sops[0].sem_flg = 0;
14 if (semop(semid, sops, 1) == -1)
15     ddErrExit("semop");

```



note

- questa soluzione al race problem non è richiesta in tutte le applicazioni.
 - e in alcune varianti di BSD è inefficace, in quanto il momento di esecuzione della `semop()` non viene registrato dal campo `sem_otime...`
 - in generale non ne abbiamo bisogno se un processo è in grado di creare e inizializzare il semaforo prima che qualsiasi altro tenti di utilizzarlo.
 - è questo il caso, per esempio, **in cui un genitore crea e inizializza il semaforo prima di creare i processi figli con cui condivide il semaforo**
 - in questi casi, è sufficiente che il genitore subito dopo la `semget()` esegua un'operazione `SETVAL` o `SETALL` con la system call `semctl()`.



operazioni sui semafori

operazioni sui semafori

```
#include <sys/types.h> /* For portability */  
#include <sys/sem.h>  
  
int semop(int semid, struct sembuf *sops, unsigned int nsops);
```

Returns 0 on success, or -1 on error

- la system call **semop()** esegue una o più **operazioni sui semafori nel set identificato da *semid*.**
 - l'argomento ***sops*** è un puntatore a un array che contiene le operazioni da eseguirsi, e
 - ***nsops*** è la dimensione dell'array (che deve contenere almeno un elemento).



sembuf: semaphore operations

```
struct sembuf {  
    unsigned short sem_num; // numero semaforo  
    short sem_op; // operazione da eseguire  
    short sem_flg; // flags operazione  
        // (IPC_NOWAIT and SEM_UNDO)  
};
```

- le **operazioni** sono **eseguite atomicamente** e **nell'ordine** in cui compaiono **nell'array**. Gli elementi dell'array sops sono strutture con queste caratteristiche:
 - il membro **sem_num** identifica, all'interno del set, il semaforo sul quale si intende effettuare l'operazione.



sembuf: semaphore operations

```
struct sembuf {  
    unsigned short sem_num;  
    // operazione da eseguire  
    short sem_op;  
    short sem_flg;  
};
```

- il membro `sem_op` specifica l'operazione da eseguire:
 - se `sem_op` è maggiore di 0, il valore di `sem_op` è aggiunto al valore del semaforo.
 - quindi altri processi in attesa di diminuire il valore del semaforo possono essere svegliati per eseguire le proprie operazioni. Il processo chiamante deve avere diritti di scrittura sul semaforo.
 - se `sem_op` è uguale a 0, il valore del semaforo è testato per vedere se attualmente è uguale a 0. Se sì, l'operazione è completata immediatamente; diversamente, la `semop()` si blocca finché il valore del semaforo diviene 0.
 - il chiamante deve avere permessi in scrittura sul semaforo.



sembuf: semaphore operations

```
struct sembuf {  
    unsigned short sem_num;  
    // operazione da eseguire  
    short sem_op;  
    short sem_flg;  
};
```

- se **sem_op** è minore di 0, decrementa il valore del semaforo del valore specificato da **sem_op**.
 - se il valore corrente del semaforo è maggiore o uguale al valore assoluto specificato da **sem_op**, l'operazione è completata immediatamente.
 - diversamente, **semop()** si blocca finché il valore del semaforo è stato aumentato tanto da permettere che l'operazione venga eseguita (senza produrre un valore negativo).
- il chiamante deve avere diritti di scrittura sul semaforo.



semantica sottostante

- l'incremento del valore di un semaforo corrisponde a **rendere disponibile una risorsa** così che altri processi possano utilizzarla;
- il decremento del valore di semaforo corrisponde **a riservare la risorsa per un uso esclusivo** da parte di questo processo.
- Il tentativo di ridurre il valore di un semaforo può restare bloccato se il valore del semaforo è troppo basso, cioè se qualche altro processo ha già ottenuto l'uso esclusivo per la risorsa in questione.



cosa accade a un processo bloccato

- quando una system call `semop()` si blocca, il processo resta bloccato finché:
 - un altro processo modifica il valore del semaforo così che la richiesta possa procedere;
 - un segnale interrompe la system call `semop()`. In questo caso la `semop()` fallisce con l'errore `EINTR`.
 - un altro processo cancella il semaforo identificato da `semid`. In questo caso, la `semop()` fallisce con l'errore `EIDRM`.



non-blocking semop()

```
struct sembuf {  
    unsigned short sem_num;  
    // operazione da eseguire  
    short sem_op;  
    short sem_flg;  
};
```

- è possibile **prevenire il blocco della semop()** nell'esecuzione di un'operazione su un dato semaforo specificando il **flag IPC_NOWAIT** nel membro **sem_flg** della struttura **sembuf**.
 - in questo caso, invece di restare bloccata, la **semop()** fallisce con l'errore **EAGAIN**.



operations on multiple semaphores

- è possibile eseguire una `semop()` per compiere operazioni su molteplici semafori in uno stesso set.
- questo gruppo di operazioni è eseguito atomicamente; cioè, o la `semop()` esegue tutte le operazioni, o si blocca fino a quando non diventa possibile eseguirle simultaneamente tutte.



operations on multiple semaphores

- consideriamo un esempio:
 - l'uso di `semop()` per eseguire operazioni su tre semafori in un set.
 - Le operazioni sui semafori 0 e 2 possono non essere in grado di procedere immediatamente, a seconda dei valori correnti dei semafori.
 - se l'operazione sul semaforo 0 non può essere eseguita immediatamente, allora nessuna delle operazioni richieste viene eseguita, e la `semop()` pone in attesa il chiamante.
 - se l'operazione sul semaforo 0 può essere eseguita immediatamente, ma non l'operazione sul semaforo 2, allora se è stato specificato il flag `IPC_NOWAIT` sul semaforo 2, nessuna delle operazioni è eseguita, e la `semop()` restituisce immediatamente con l'errore `EAGAIN`.



```

struct sembuf sops[3];

sops[0].sem_num = 0;
sops[0].sem_op = -1; // DECREMENTO di 1 il semaforo 0
sops[0].sem_flg = 0;

sops[1].sem_num = 1;
sops[1].sem_op = 2; // INCREMENTO di 2 il semaforo 1
sops[1].sem_flg = 0;

sops[2].sem_num = 2;
sops[2].sem_op = 0; // ATTESA che il semaforo 2 valga 0
sops[2].sem_flg = IPC_NOWAIT; /* operazione NON SOSPENSIVA: se
l'operazione non può essere effettuata, NON attendere */

if (semop(semid, sops, 3) == -1) {
    if (errno == EAGAIN) /* il semaforo 2 si sarebbe bloccato; invece
grazie a IPC_NOWAIT ha restituito EAGAIN */
        printf("Operation would have blocked\n");
else
    errExit("semop"); // si è verificato qualche altro errore

```



gestione di blocchi multipli

l'ordine in cui vengono soddisfatte le richieste

- if multiple processes are blocked trying to decrease the value of a semaphore by the same amount, then it is indeterminate which process will be permitted to perform the operation first when it becomes possible.
- on the other hand, if processes are blocked trying to decrease a semaphore value by different amounts, then the requests are served in the order in which they become possible.



starvation (by decrease amount)

- suppose that a semaphore currently has the value 0, and process A requests to decrease the semaphore's value by 2, and then process B requests to decrease the value by 1.
- if a third process then adds 1 to the semaphore, process B would be the first to unblock and perform its operation, even though process A was the first to request an operation against the semaphore.
- in poorly designed applications, such scenarios can lead to starvation, whereby a process remains blocked forever because the state of the semaphore is never such that the requested operation proceeds.
- also in case multiple processes adjust the semaphore in such a way that its value is never more than 1, process A remains blocked forever.



starvation (by multiple sem operations)

- starvation can also occur if a **process is blocked trying to perform operations on multiple semaphores**. consider the following example, with a pair of semaphores, both of which initially have the value 0:
 1. process A makes a request to subtract 1 from semaphores 0 and 1 (blocks).
 2. process B makes a request to subtract 1 from semaphore 0 (blocks).
 3. process C adds 1 to semaphore 0.
- at this point, process B unblocks and completes its request, even though it placed its request later than process A.
 - again, it is possible to devise scenarios in which process A is starved while other processes adjust and block on the values of the individual semaphores.



Implementing a Binary Semaphores Protocol

complessità e semplificazione

- la API per i semafori di System V semaphores è complessa,
 - perché il valore dei semafori può essere modificato di quantità arbitrarie,
 - perché i semafori sono allocati in set di semafori, e le operazioni sono eseguite su set di semafori.
- entrambe queste caratteristiche forniscono **funzionalità più estese di quelle tipicamente necessarie**, quindi è utile implementare protocolli più semplici.



binary semaphores

- un protocollo largamente utilizzato è quello dei **semafori binari**. Un semaforo binario ha due valori: **available** (libero) e **reserved** (in uso). sui semafori binari sono definite due operazioni:
 - **Reserve (wait, o P)**: Tenta di riservare questo semaforo per uso esclusivo.
 - se il semaforo è già stato riservato da un altro processo, l'operazione si blocca fino a quando il semaforo è rilasciato.
 - **Release (signal, o V)**: Libera un semaforo correntemente riservato, così che possa essere riservato da un altro processo.



- un modo abituale per rappresentare questi stati è utilizzare il valore 1 per indicare free e il valore 0 per riservato, con le operazioni reserve e release: l'una decrementa di uno il valore del semaforo, l'altra lo incrementa di uno.



implementazione

- tutte le funzioni in questa implementazione hanno **due argomenti**, che identificano **un set di semafori** e **il numero di un semaforo all'interno di quel set**.

```
int initSemAvailable(int semId, int semNum);  
int initSemInUse(int semId, int semNum);  
int reserveSem(int semId, int semNum);  
int releaseSem(int semId, int semNum);
```



```
// Initialize semaphore to 1 (i.e., "available")
int initSemAvailable(int semId, int semNum) {
    union semun arg;

    arg.val = 1;
    return semctl(semId, semNum, SETVAL, arg);
}

// Initialize semaphore to 0 (i.e., "in use")
int initSemInUse(int semId, int semNum) {
    union semun arg;

    arg.val = 0;
    return semctl(semId, semNum, SETVAL, arg);
}
```



// Reserve semaphore - decrement it by 1

```
int reserveSem(int semId, int semNum) {
    struct sembuf sops;

    sops.sem_num = semNum;
    sops.sem_op = -1;
    sops.sem_flg = 0;

    return semop(semId, &sops, 1);
}
```

// Release semaphore - increment it by 1

```
int releaseSem(int semId, int semNum) {
    struct sembuf sops;

    sops.sem_num = semNum;
    sops.sem_op = 1;
    sops.sem_flg = 0;

    return semop(semId, &sops, 1);
}
```

