Meccanismi di sincronizzazione: i semafori

Il *semaforo* é un *tipo di dato astratto* con associate le operazioni (atomiche) che in pseudo-C possiamo descrivere così:

- init(semaphore *sp, int val_ini); *(da utilizzare solo una volta, quando si crea il semaforo)*
- up(semaphore *sp);
- down(semaphore *sp);

Indichiamo con *s.val* il valore (intero non negativo) di un semaforo *s*, che va inizializzato con *init*

Poi il valore si può modificare solo attraverso le procedure *up* e *down* che si comportano nel modo seguente (definizione informale):

- *down*: se *s.val > 0*, tale valore viene decrementato, altrimenti il processo/thread che esegue l'operazione deve *attendere*
- *up:* se vi sono processi/threads in attesa per effetto di una *down*, uno di questi termina l'attesa e conclude l'esecuzione della *down*, altrimenti *s.val* viene incrementato

up e down devono essere OPERAZIONI ATOMICHE
(indivisibili : sono esse stesse sezioni critiche)

Attenzione alla notazione: Up e Down

In alcuni esempi si può incontrare la notazione s.up() e s.down() – mutuata dai linguaggi ad oggetti

Altri nomi usati in alcuni testi per queste funzioni sono:

- signal-wait
- V e P, dall'olandese *Verhogen=incrementare* e *Proberen=verificare* (o altre parole olandesi)

perché i semafori sono stati inventati da Edsger W. Dijkstra

Semafori e mutua esclusione

Per garantire l'esecuzione in mutua esclusione di sezioni critiche si può utilizzare un semaforo *s*, inizializzato a 1 e condiviso da tutti i processi/thread che contengono sezioni critiche relative a determinate strutture dati (per esempio al vettore CC[ ] dell'esempio VersaSulConto):

```
down(&s);
Sezione Critica
up(&s);
```

verde per l'ingresso in sezione critica: vale 1

rosso: vale 0.

Un processo che vuole entrare in sezione critica (down(&s)) :

- se trova il semaforo verde, lo imposta a rosso ed entra;
- se lo trova rosso viene sospeso

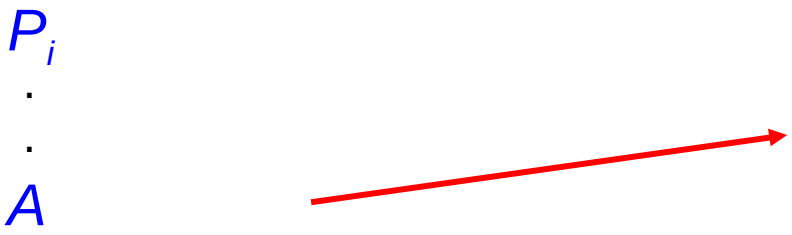
Quando il processo esce dalla sezione critica (up) il semaforo torna verde se non ci sono processi in attesa; o se ve ne sono, uno può entrare

Mutua esclusione in VersaSulConto con semafori

P1	sem_cc	CC[1200]	P2
	1	2.000	
			3.
1. down(&sem_cc);	0		7. down(&sem_cc);
2. Saldo = CC[1200];			8. Saldo = CC[1200];
		P1: Saldo = 2.000	*P2: Saldo = 2.200*
4. Saldo = Saldo + 200;			9. Saldo = Saldo + 350;
		P1: Saldo = 2.200	*P2: Saldo = 2.550*
5. CC[1200] = Saldo;			10. CC[1200] = Saldo;
		P1: CC[1200] = 2.200	
		2.200	
6 up(&sem_cc);	0		11. up(&sem_cc);
			P2: CC[1200] = 2.550
	1	2.550	

Altri tipi di sincronizzazione

- Si vuole eseguire *B* in P_k solo **dopo** che *A* è stato eseguito in P_i



- Usiamo un semaforo *flag* inizializzato a 0
- Codice:

P_i	P_k
A	*down*(&flag)
up(&*flag*)	*B*

I semafori: implementazione

Per una possibile realizzazione come chiamate di sistema, indichiamo con s.queue la lista di processi/thread associata a s

```
down(&s) :  if  (s.val ==0)
            {  inserisci il processo corrente p in s.queue;
               cambia stato di p a bloccato;
               scheduler(); dispatcher();
            }
            else s.val--

up(&s) :    if (s.queue non vuota)
            { estrai un processo p da s.queue;
              cambia stato di p a pronto;
            }
            else
             s.val++ ;
```

non è detto che s.queue sia gestita First-In-First-Out

ma è il modo più semplice per garantire *attesa limitata,* e il più neutrale; non sapendo per cosa viene usato il semaforo, non ha senso realizzare una *politica* dentro a up()

I semafori: implementazione

Per una possibile realizzazione come chiamate di sistema, indichiamo con s.queue la lista di processi/thread associata a s

```
down(&s) :  if  (s.val ==0)
            {  inserisci il processo corrente p in s.queue;
               cambia stato di p a bloccato;
               scheduler(); dispatcher();
            }
            else s.val--
```

down indivisibile : evita ad es. che con s.val==1 due processi «vedano» s.val>0 e procedano (violando *mutua esclusione)*

```
up(&s) :    if (s.queue non vuota)
            { estrai un processo p da s.queue;
              cambia stato di p a pronto;
            }
            else
              s.val++ ;
```

se non indivisibili, cosa potrebbe accadere con due up?
Ad es., estraggono e mettono pronto lo stesso processo, un altro rimane inutilmente sospeso (viola *progresso*)

Semafori come system call

La up() e la down() potrebbero essere implementate all'interno del sistema operativo, come system call

Per garantire che esse vengano eseguite in modo atomico, su un sistema uniprocessore si può usare la tecnica della disabilitazione degli interrupt (che in questo caso verrebbe usata esclusivamente dal S.O., non data in mano al programmatore)

Su un sistema multiprocessore, si può usare la soluzione al problema della mutua esclusione basato sulla istruzione TSL
L'attesa attiva in questo caso può essere accettabile poiché la sezione critica è molto breve: se i processi/thread girano su processori diversi, uno fa attesa attiva al massimo per il tempo necessario agli altri per eseguire il codice della up o della down, che consiste in poche istruzioni, a differenza di quello di una sezione critica che inserisce chi scrive i programmi che usano up/down, la cui durata non è limitata

Nota su semafori e programmazione

Negli esempi tratti dal testo i semafori sono dichiarati così:

typedef int semaphore;
semaphore mutex = 1;

ma la notazione è usata solo *per analogia* con il C:
è vero che come indica un commento “i semafori sono un tipo speciale di interi”, ma il loro essere speciali consiste nell’essere strutture che devono essere messe a disposizione dal sistema operativo, insieme con le operazioni predefinite (down e up) realizzate attraverso chiamate di sistema

Inoltre è opportuno che *non* si possa operare in altro modo sui semafori, ad esempio utilizzando il nome della variabile in istruzioni di assegnazione (vedasi il concetto di “tipo di dato astratto” e di “oggetto” nella programmazione)

Sono stati realizzati (ma poco usati) dei linguaggi per la programmazione concorrente in cui esiste veramente un tipo “semaphore” e il compilatore interagisce con il sistema operativo

Semafori binari o semafori contatore

Un semaforo binario può assumere solo i valori 0 o 1 (o si tratta di un semaforo generale, usato solo in modo che assuma tali valori)

Ha le stesse operazioni già viste; ma nel caso si tratti di un vero e proprio semaforo binario (non un semaforo generale usato come binario) se si esegue una up() quando s.val è già 1, questa non ha alcun effetto (oppure, come è comodo dire negli standard: l'effetto è indefinito, cioè: meglio scrivere i programmi in modo che non succeda, perché le conseguenze sono a scelta dell'implementazione)

Possono essere utilizzati per garantire la mutua esclusione, inizializzandoli a 1

Se usati solo a tale scopo possono essere chiamati *mutex*, eventualmente con inizializzazione implicita

Ma è un semaforo binario anche quello usato per "*B* in P_k solo dopo *A* in P_i"

Va inizializzato a 0

Semafori binari o semafori contatore

Un semaforo contatore può assumere qualsiasi valore ≥ 0. Può essere usato ad esempio per il problema di sincronizzazione con un numero N di risorse da assegnare:

si inizializza a N, numero di “risorse” (in senso lato) disponibili

preleva = down(&s)

rilascia = up(&s)

N processi/threads possono superare down senza nessuna up, l’N+1 esimo viene sospeso

In generale, in ogni momento s.val è ≥ 0, e vale:

s.val = N – numero down completate + numero up completate

cioè, intendendolo come numero di “risorse” disponibili:

risorse totali – risorse prelevate + risorse rilasciate

Soluzione del problema P-C usando i semafori

```
1  #define N 100           / * number of slots in the buffer * /
2  typedef int semaphore;  / * semaphores are a special kind of int * /
3  semaphore mutex = 1;    / * controls access to critical region * /
4  semaphore empty = N;    / * counts empty buffer slots * /
5  semaphore full = 0;     / * counts full buffer slots * /
6
7  void producer(void)
8  {
9      int item;
10     while(TRUE) {                    / * TRUE is the constant 1 * /
11         item = produce item( );      / * generate something to put in buffer * /
12         down(&empty);                / * decrement empty count * /
13         down(&mutex);                / * enter critical region * /
14         inser t item(item);          / * put new item in buffer * /
15         up(&mutex);                  / * leave critical region * /
16         up(&full);                   / * increment count of full slots * /
17     }
18 }
19 void consumer(void)
20 {
21     int item;
22     while (TRUE) {                       / * infinite loop * /
23         down(&full);                     / * decrement full count * /
24         down(&mutex);                    / * enter critical region * /
25         item = remove item( );           / * take item from buffer * /
26         up(&mutex);                      / * leave critical region * /
27         up(&empty);                      / * increment count of empty slots * /
28         consume item(item);              / * do something with the item * /
29     }
30 }
```

semaforo binario

} semafori contatore

sospensiva se empty=0 (buffer pieno)

oppure risveglia consumatore

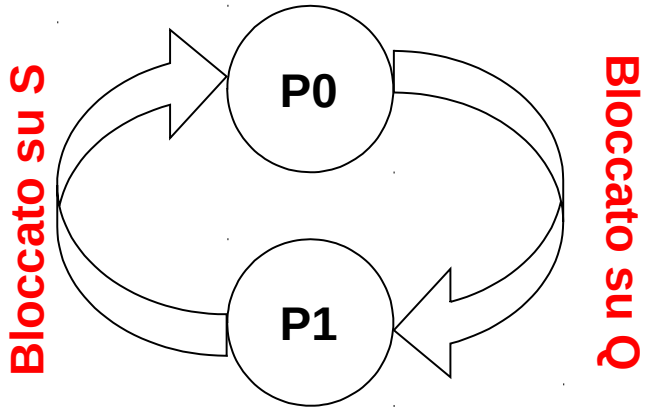
sospensiva se full=0 (buffer vuoto)

oppure risveglia produttore

Problemi di sincronizzazione: il deadlock (stallo)

- **Deadlock** (stallo) – si verifica quando due o più processi attendono indefinitamente il verificarsi di un evento che può essere causato solo da uno dei processi stessi
  - Es.: siano *S* e *Q* due semafori inizializzati a 1

P_0	P_1
1. *down(&S);*	**2.** *down(&Q);*
4. *down (&Q);*	**3.** *down(&S);*
:	:
up(&S);	*up(&Q);*
up(&Q);	*up(&S);*

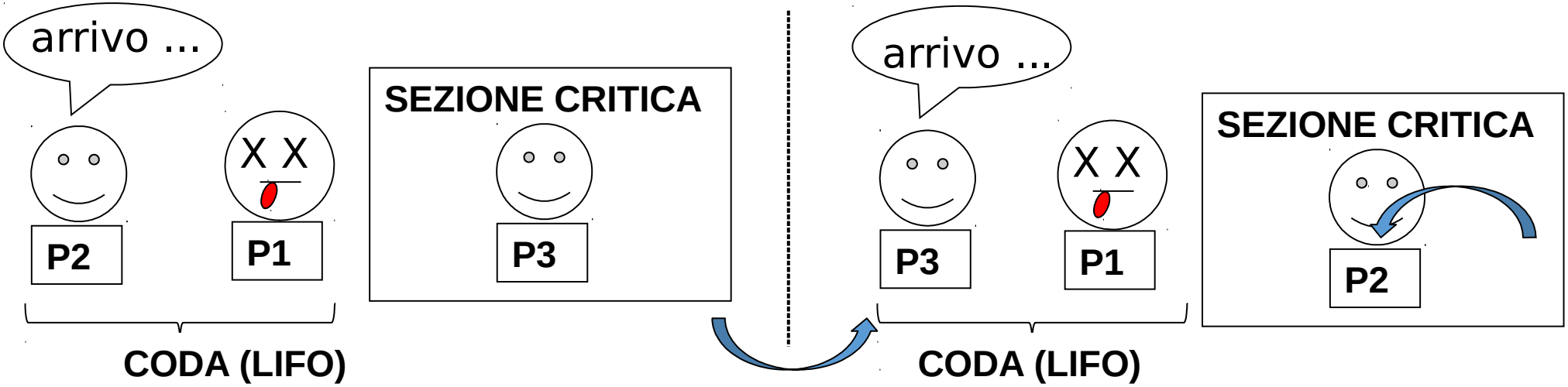


Problemi di sincronizzazione: la starvation

- Starvation (morte di fame) – si verifica quando un processo P rimane per sempre in una coda d'attesa (per es. di un semaforo) perché altri processi vengono ripetutamente risvegliati prima di P.

Es, 3 (o più) processi usano un semaforo per la mutua esclusione. Se la coda del semaforo viene gestita con politica FIFO (First In First Out), l'attesa di un processo in coda al semaforo è certamente limitata.

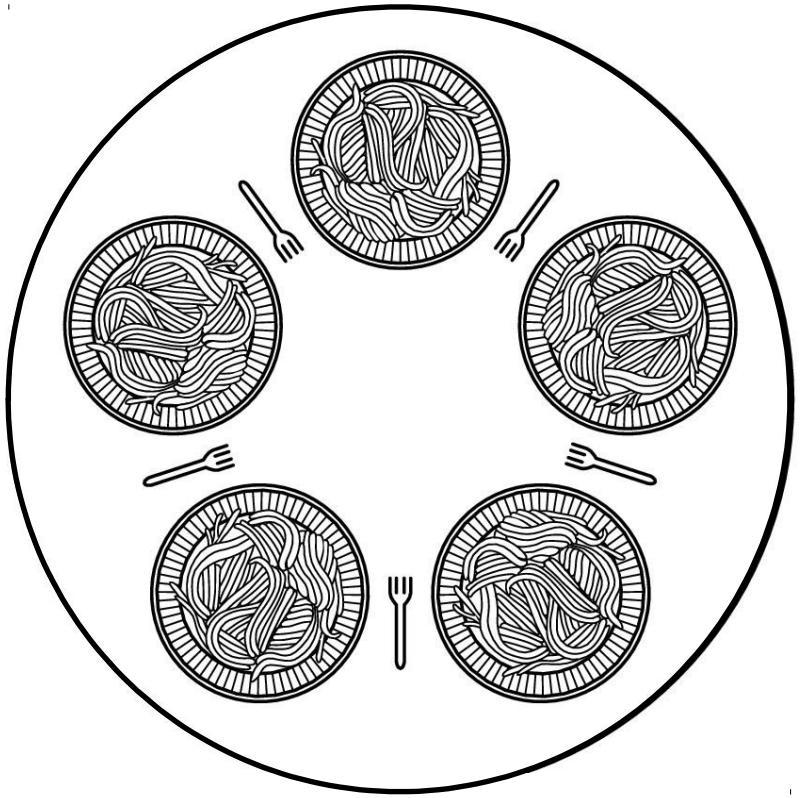
Se la coda venisse gestita con politica LIFO (Last In First Out), è possibile che un processo rimanga in coda all'infinito (es.: P1 è in coda, P3 è in s.c., P2 arriva in coda, quando P3 lascia la s.c., P2 passa avanti a P1; se poi P3 arriva in coda prima che P2 lasci la s.c., P3 passa avanti a P1, ecc... all'infinito)



Dining Philosophers

- I filosofi pensano, ma ogni tanto mangiano;
- per mangiare necessitano di 2 forchette, quella alla propria sinistra e quella alla propria destra

Rappresenta il caso di vari processi che per una parte del loro codice (pensare) non devono sincronizzarsi; per un'altra (mangiare) devono utilizzare più risorse in mutua esclusione



Dining Philosophers

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
      while (TRUE) {
            think( );                           /* philosopher is thinking */
            take_fork(i);                       /* take left fork */
            take_fork((i+1) % N);               /* take right fork; % is modulo operator */
            eat( );                             /* yum-yum, spaghetti */
            put_fork(i);                        /* put left fork back on the table */
            put_fork((i+1) % N);                /* put right fork back on the table */
      }
}
```

se dichiariamo un array di semafori *fork[N]*
take_fork(i) potrebbe essere semplicemente *down(&fork[i])* ,
e la *put_fork(i)* una *up.*

Potrebbe?

Dining Philosophers

Può portare ad una situazione di deadlock: se tutti i filosofi prelevano la forchetta di sinistra e poi si sospendono attendendo di acquisire la forchetta di destra, rimarranno bloccati in questo stato indefinitamente

Riprendiamo la definizione di deadlock:

un insieme di processi in attesa di un evento che può essere provocato solo da un processo nell'insieme stesso, in questo caso

Il filosofo 0 supera la down(&fork[0]) ma rimane sospeso sulla down(&fork[1]),

il filosofo 1 supera la down(&fork[1]) ma rimane sospeso sulla down(&fork[2]),

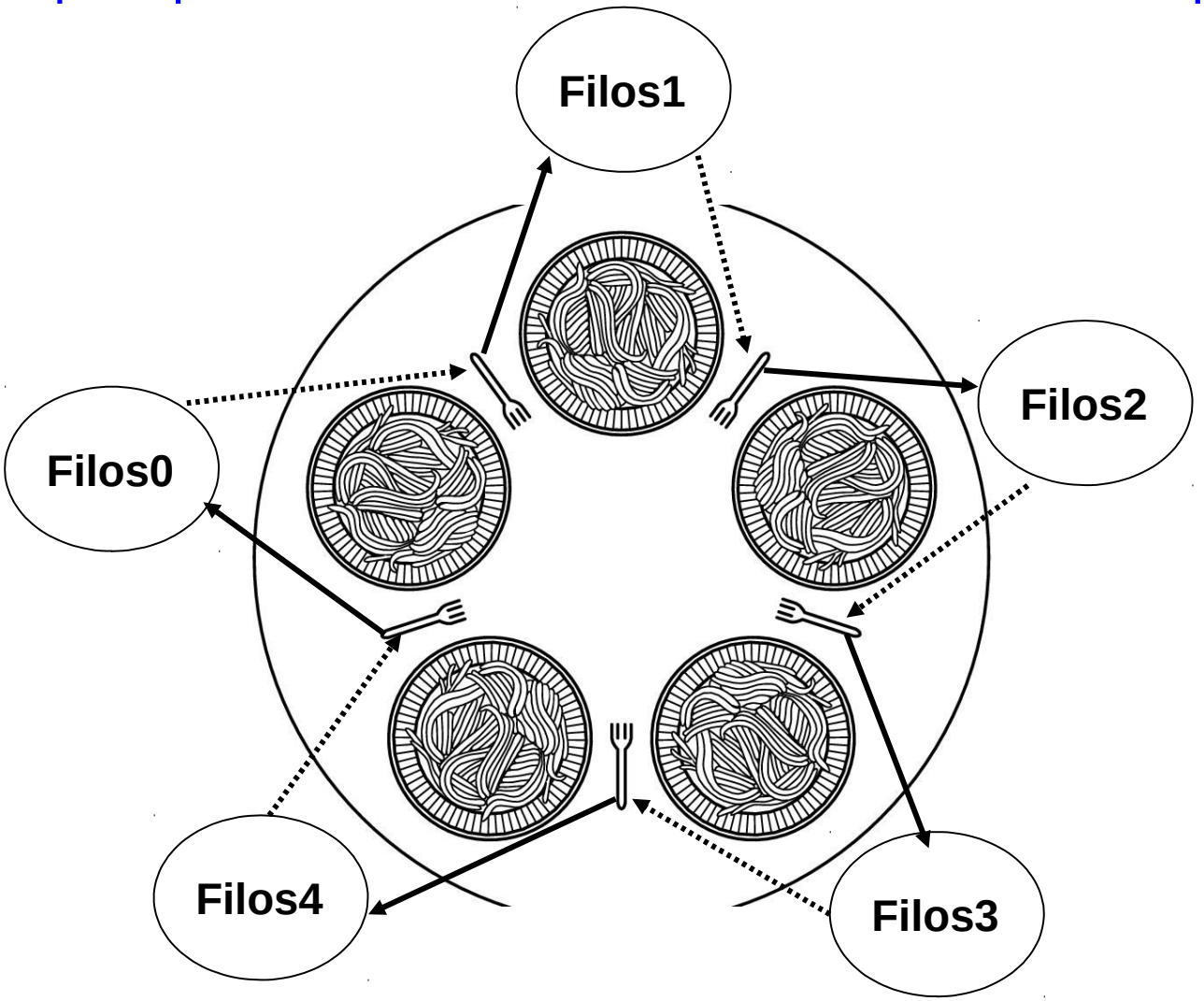
...,

il filosofo N supera la down(&fork[N]) ma rimane sospeso sulla down(&fork[0])

⇒ Ciclo di N processi in attesa

Dining Philosophers

il filosofo 0 attende la forchetta che può essere liberata solo dal filosofo 1, il quale attende la forchetta che può essere liberata solo dal filosofo 2, … il filosofo 4, il quale attende la forchetta che può essere liberata solo dal filosofo 0. I cinque processi formano un ciclo di attesa dal quale non si può uscire



Dining Philosophers

Una possibile soluzione consiste nel cambiare l'ordine di acquisizione delle forchette ad uno dei filosofi: per esempio il filosofo 4 potrebbe acquisire prima la forchetta di destra (0) e poi quella di sinistra (4)

Un'altra possibile soluzione consiste nell'ammettere al più 4 filosofi nella fase di acquisizione forchette, utilizzando un semaforo contatore inizializzato a 4, e aggiungendo down(&count) prima dell'acquisizione delle forchette ed una up(&count) subito dopo l'acquisizione di entrambe le forchette

Una terza soluzione possibile consiste nell'acquisire le risorse (forchette) contemporaneamente, anziché una alla volta. Usiamo un array di *N* semafori *s[N]*. Quando il processo *i* non può acquisire le forchette (perché non sono entrambe disponibili) si sospende sul semaforo *s[i]*. Quando un semaforo viene usato in questo modo, si chiama *semaforo privato*

Politiche di assegnazione risorse tramite semafori privati

I semafori ***privati*** sono tali solo per come vengono usati: il meccanismo messo a disposizione dalle funzioni è lo stesso, senza alcun controllo su quale processo/thread usa i semafori e come

- Un semaforo privato s_priv_P «di» un processo P (o di una classe di processi) è inizializzato a 0
- Solo il processo P (o un processo della classe associata al semaforo) esegue down(&s_priv_P); la esegue quando deve attendere che diventi vera una condizione (booleana) di sincronizzazione
- qualsiasi processo (P incluso) può eseguire up(&s_priv_P) se serve svegliare P, o serve non farlo sospendere se fa down, perché è vera la condizione di sincronizzazione

Dining Philosophers: soluzione tramite semafori privati

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N−1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N−1 */
{
      while (TRUE) {                        /* repeat forever */
            think( );                       /* philosopher is thinking */
            take_forks(i);                  /* acquire two forks or block */
            eat( );                         /* yum-yum, spaghetti */
            put_forks(i);                   /* put both forks back on table */
      }
}
```

Dining Philosophers: soluzione tramite semafori privati

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                     /* enter critical region */
      state[i] = HUNGRY;                /* record fact that philosopher i is hungry */
      test(i);                          /* try to acquire 2 forks */
      up(&mutex);                       /* exit critical region */
      down(&s[i]);                      /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                     /* enter critical region */
      state[i] = THINKING;              /* philosopher has finished eating */
      test(LEFT);                       /* see if left neighbor can now eat */
      test(RIGHT);                      /* see if right neighbor can now eat */
      up(&mutex);                       /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

"ovviamente" down(&s[i]) è dopo up(&mutex)

per non fermarsi quando fa down, in take_forks;
o per svegliare il vicino, in put_forks;

Dining Philosophers: soluzione tramite semafori privati

La soluzione può essere elaborata per garantire l'assenza di *starvation*: per esempio non permettendo ad un filosofo di mangiare più di *k* volte di fila se un suo vicino è affamato:

- ogni volta che il filosofo *i* preleva la forchetta che serve anche a un vicino affamato si incrementa un contatore
- quando il contatore arriva a *k* il filosofo si sospende