Semafori

Index

Introduction

I semafori sono delle particolari strutture dati su cui si possono fare tre operazioni *atomiche*:

- `initialize`

- `decrement` (o `semWait`) → può mettere il processo in blocked: niente CPU sprecata come il busy waiting
- `increment` (o `semSignal`) → può mettere un processo blocked in ready

Si tratta di syscall, quindi sono eseguite in kernel mode e possono agire direttamente sui processi

Pseudocodice

```
struct semaphore {
        int count;
        queueType queue;
};
void semWait(semaphore a) {
        s.count--;
        if (s.count < 0) {
                /* place this process in s.queue */;
                /* block this process */
        }
}
void semSignal(semaphore a) {
        s.count++;
        if (s.count <= 0) {
                /* remove a process P from s.queue */;
                /* place process P on ready list */;
        }
}
```

In questo caso `count` corrisponde al numero di processi che si trovano nella queue

Semafori binari - pseudocodice

```
struct binary_semaphore {
        enum {zero, one} value;
        queueType queue;
};
void semWait(binary_semaphore a) {
        if (s.value == one)
                s.value = zero;
        else {
                /* place this process in s.queue */;
                /* block this process */;
        }
}
```

```
void semSignalB(binary_semaphore a) {
        if (s.queue is empty())
                s.value = one;
        else {
                /* remove a process P from s.queue */;
                /* place process P on ready list */;
        }
}
```

Semafori - pseudo codice “vero”

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts */;
   }
   else
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

Errore nel codice di sinistra: manca un else prima di s.flag $= 0$

Consideriamo tre processi A, B e C

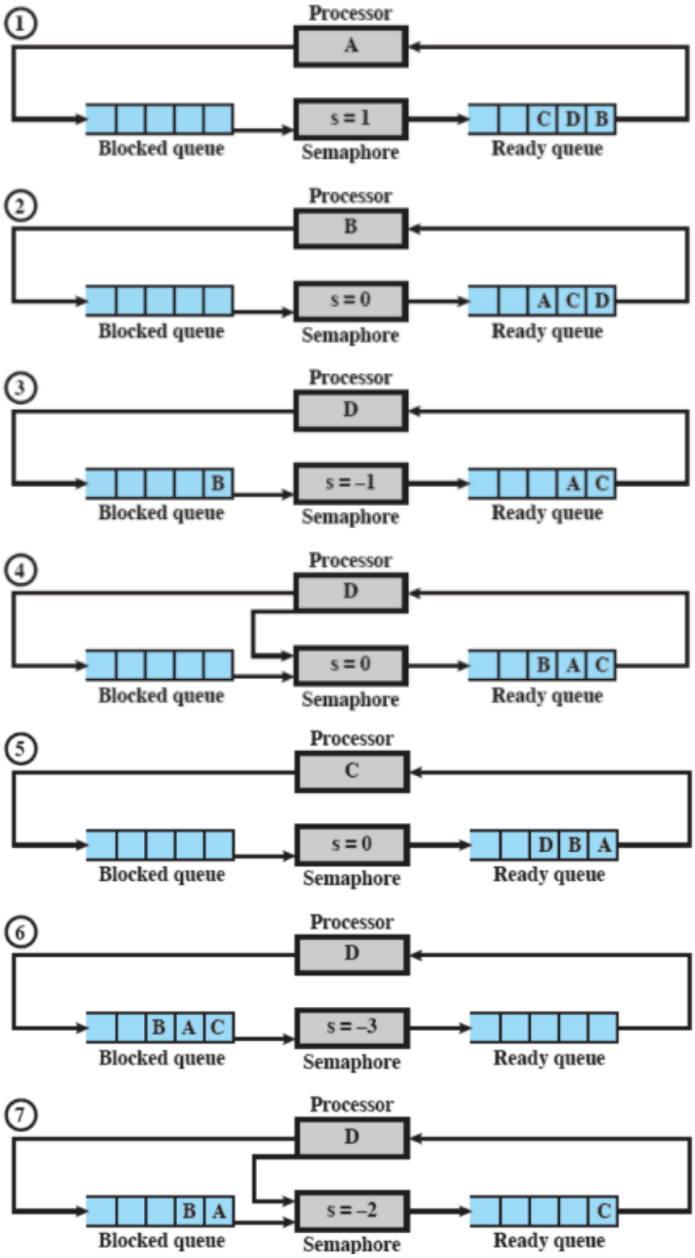
1. A ha già completato la `semWait` e sta eseguendo codice in sezione critica. `s.count` è 0
2. B entra in `semWait`, count va a -1 e B diventa blocked. `s.flag = 0`
3. tocca ad A, che esegue sezione critica e `semSignal`. `s.count = 0`
   - il sistema dunque sposta B che era in wait sul semaforo da blocked a ready
   - A completa `semSignal`. `s.flag = 0`
4. C entra in `semWait`, passa il while `compare_and_swap` e viene interrotto immediatamente dallo scheduler. `s.flag = 1`

5. B riprende l'esecuzione, imposta `s.flag = 0`, esegue la sua sezione critica e chiama `semSignal`. Passa il while. `s.flag = 1` imposta `s.count = 1`, termina `semSignal` ed imposta `s.flag = 0`
   Stato corrente: C fermo prima di `s.count--`; `s.count == 1`; `s.flag == 0`
6. Arriva un nuovo processo D, che entra in `semWait`. Passa il while, ora `s.flag = 1`
7. D esegue `s.count--`: legge `s.count` da memoria e lo porta in `eax`. `eax == 1` scheduler interrompe D ed esegue C
8. C continua da dov'era: esegue `s.count--`, che diventa ora 0, quindi non va in block
9. C preempted nella sezione critica
10. tocca a D, che continua il calcolo di prima: salva `eax - 1` in `s.count`, che rimane 0
11. D non va in block, e continua nella sezione critica
    Race condition

Semafori deboli e forti

In base a come scelgo il processo da sbloccare all'interno della coda dei processi si parla di **semafori deboli** e **semafori forti**
I semafori forti sono quelli che usano la politica FIFO (è una coda, manda in esecuzione il processo che aspetta da più tempo)
Ci stanno però sistemi operativi che usano i semafori cosiddetti "deboli" per cui una politica non è specificata, uno qualsiasi dei processi in coda viene sbloccato ma non è dato sapere quale

Semafori forti - esempio

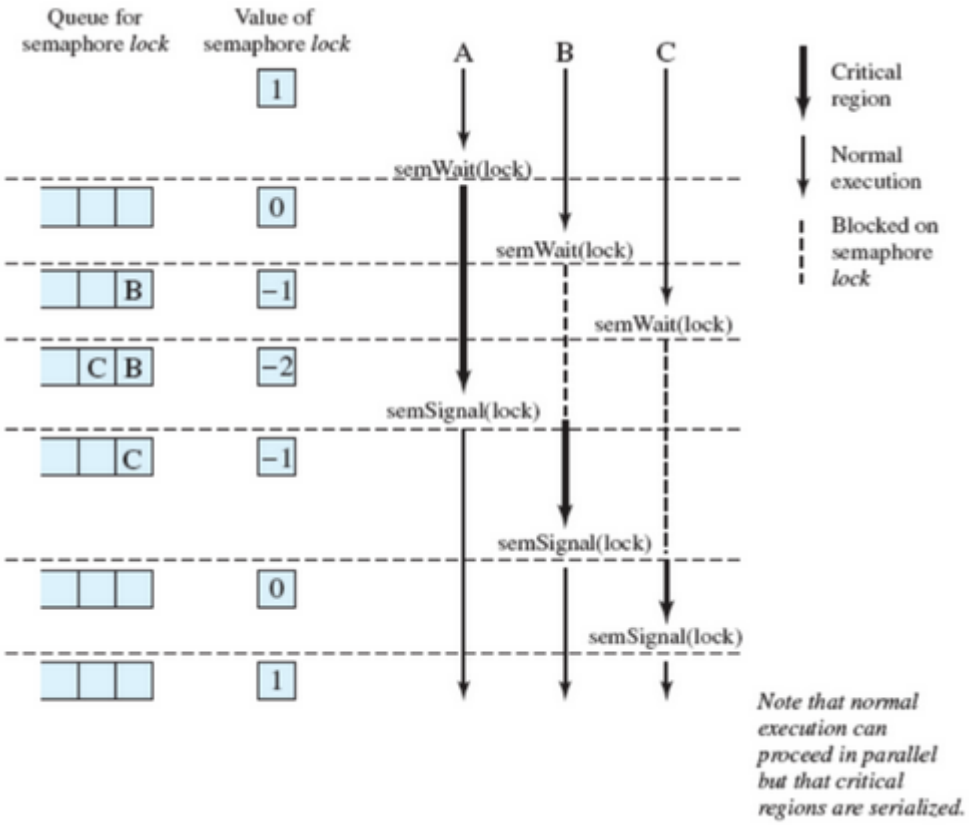


Mutua esclusione con i semafori

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;

void P(int i) {
        while (true) {
                semWait(s);
                /* critical section */
                semSignal(s);
                /* remainder */
        }
}
```

```
void main() {
        parbegin(P(1), P(2), ..., P(n));
}
```

In questo caso non si ha starvation (a meno che i semafori non siano deboli)



Problema del produttore/consumatore

Oltre al problema della mutua esclusione ci sono alcuni problemi che riguardano i processi che cooperano come il problema del **produttore/consumatore**

Come situazione generale si ha un processo che è produttore, che crea dati e li deve mettere in un buffer e ci sta un consumatore, che prende i dati dal buffer uno alla volta (per farci calcoli). In ogni caso al buffer ci può accedere un solo processo (sia esso produttore o consumatore)

Il problema sta principale (oltre a dover garantire la mutua esclusione) sta nel garantire che il produttore non inserisca dati se il buffer è già pieno e che il consumatore non prenda dati quando il buffer è vuoto

Pseudocodici

Per questo primo esempio, facciamo finta che il buffer sia infinito (non consideriamo il problema di buffer pieno)

> **Info**
>
> Non si creano problemi se contemporaneamente si sta producendo e consumando

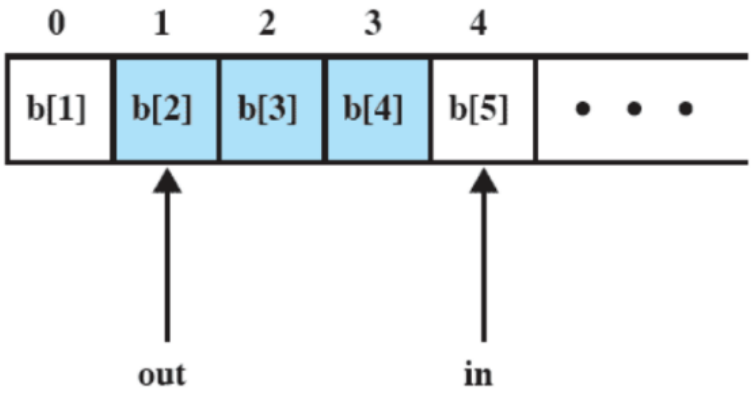
`b` rappresenta il buffer in cui viene inserito il nuovo elemento `v`

```
while (true) {
        /* produce item v */
        b[in] = v;
        in++;
}
```

se `out` è più grande di `in` (variabile globale) vuol dire che ho consumato tutto il buffer e rimane quindi in active wait

```
while (true) {
        while (in <= out) /* do nothing */;
        w = b[out];
        out++;
        /* consume item w */
}
```

Il buffer



Soluzione sbagliata

Vediamo un esempio di soluzione (sbagliata) usando i semafori binari

```
/* program producerconsumer */
int n; // numero elemnti buffer
binary_semaphore s = 1, delay = 0;

void producer() {
        while (true) {
                produce();
                semWaitB(s);
                append();
                n++;
                if(n == 1) semSignalB(delay);
                semSignalB(s);
        }
}

void consumer() {
        semWaitB(delay);
        while (true) {
                semWaitB(s);
                take();
                n--;
                m = n;
                semSignalB(s);
                consume();
                if(n == 0) semWaitB(delay);
        }
}

void main() {
        n = 0;
        parbegin(producer, consumer);
}
```

Possibile scenario

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semiSignlaB(s)	1	-1	0

Si hanno problemi nel caso in cui venga mandato in esecuzione il produttore prima che il consumatore faccia il `consume()`.
In questo caso infatti, se lo scheduler mandasse in esecuzione due volte il consumer ci si ritroverebbe in una situazione in cui `delay` è 1 (si potrebbe iniziare a consumare) ma `n` è 0 e nonostante ciò ci è permessa un'operazione di `take()` (`n` arriva addirittura ad essere -1)
Sostanzialmente il problema è stato che il non è stata eseguita la prima `semWaitB(delay)` dopo il `consume()` in quanto è stato modificato n

Soluzione corretta

Nella soluzione seguente salviamo il valore `n` nella variabile `m` così non ho problemi anche in caso di modifiche su `n`

```
/* program producerconsumer */
int n; // numero elemnti buffer
binary_semaphore s = 1, delay = 0;

void producer() {
        while (true) {
                produce();
                semWaitB(s);
                append();
                n++;
                if(n == 1) semSignalB(delay);
                semSignalB(s);
```

```
        }
}

void consumer() {
        int m; /* a local variable */
        semWaitB(delay);
        while (true) {
                semWaitB(s);
                take();
                n--;
                m = n;
                semSignalB(s);
                consume();
                if(m == 0) semWaitB(delay);
        }
}

void main() {
        n = 0;
        parbegin(producer, consumer);
}
```

Soluzione con semafori generali

Utilizzando i semafori generali non ho più la necessità di utilizzare la variabile `n` per tenere il conto degli elementi nel buffer in quanto posso contare sui contatori dei semafori

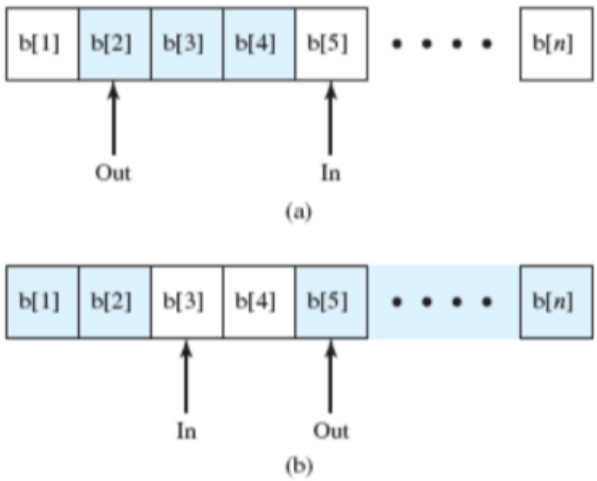
```
/* program producerconsumer */
semaphore n = 0, s = 1;

void producer() {
        while (true) {
                produce();
                semWait(s);
                append();
                semSignal(s);
                semSignal(n);
        }
}

void consumer() {
        while (true) {
                semWait(n);
                semWait(s);
```

```
                take();
                semSignalB(s);
                consume();
        }
}

void main() {
        parbegin(producer, consumer);
}
```

Produttori e consumatori con buffer circolare

Torniamo adesso ad un caso reale, ovvero quello in cui il buffer non sia infinito. Ciò si realizza attraverso un **buffer circolare**

Block on	Unblock on
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed



Buffer circolare: pseudocodici

La dimensione effettiva del buffer è $n - 1$ (altrimenti non si potrebbe capire se `in == out` implichi un buffer pieno o vuoto)

```
while (true) {
        /* produce item v */
        while ((in+1)%n == out) /* do nothing*/; // test pieno
        b[in] = v;
```

```
        in = (in+1)%n;
}
```

```
while (true) {
        while (in == out) /* do nothing*/; // test vuoto
        w = b[out];
        out = (out+1)%n;
        /* consume item w */
}
```

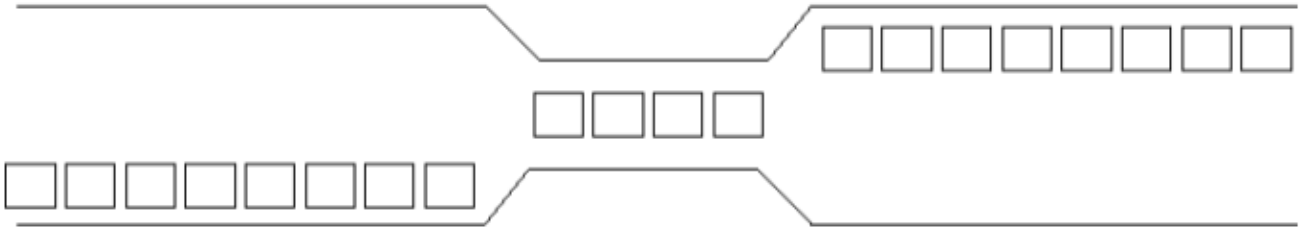
Soluzione generale con i semafori

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore n = 0, s = 1, e = sizeofbuffer;

void producer() {
        while (true) {
                produce();
                semWait(e); // viene decrementato ogni volta finché
non si arriva
                semWait(s); // fino a 0 quando il buffer è pieno
                append();
                semSignal(s);
                semSignal(n);
        }
}

void consumer() {
        while (true) {
                semWait(n);
                semWait(s);
                take();
                semSignalB(s);
                semSignalB(e); // viene incrementato e "sveglia" il
produttore
                consume();
        }
}

void main() {
        parbegin(producer, consumer);
}
```

Esempi

Analizziamo questo esempio che ci permette di comprendere appieno la potenzialità dei semafori

Trastevere



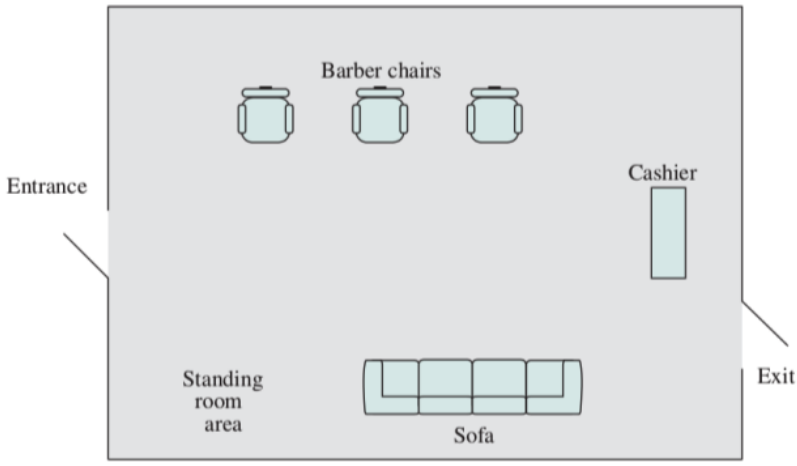
I blocchetti sono le macchine, mentre il tubo è una strada di Trastevere.
La strada si restringe in un senso unico alternato (massimo 4 auto alla volta). Vince chi arriva prima e non si può essere pareggio
Assumendo semafori *strong*, le macchine dovrebbero impiegare la strettoia nell'ordine con cui arrivano (o si accodano dalla propria parte)

```
semaphore z = 1;
semaphore strettoia = 4;
semaphore sx = 1;
semaphore dx = 1;
// incrementando nsx o ndx in due processi diversi me lo potrei
ritrovare incrementato solo di 1
int nsx = 0; // num auto sx, variabile globale (race condition)
int ndx = 0; // numero auto dx, variabile globale (race condition)

macchina_dal_lato_sinistro () {
        // chi deve passare per primo
        wait(z); // impedisce che la funzione da dx e da sx possano
essere
             // eseguite in contemporanea
        wait(sx);   // inizio sezione critica
        nsx++;
        if(nsx == 1)  // se sono il primo ad essere arrivato
                wait(dx); // blocco le auto arrivate dall'altro lato
        signal(sx); // fine sezione critica
        signal(z);

        // se ero l'ultimo ora tocca all'altro lato passare
        wait(strettoia);
        passa_strettoia();
        signal(strettoia);
        wait(sx);
```

```
        nsx--;
        if(nsx == 0)    // se ero l'ultimo
                signal(dx); // le macchine dall'altro lato possono
passare
        signal(sx);
}
```

Nella versione `macchina_dal_lato_destro` cambia solamente il fatto che tutti i `sx` diventano `dx` e viceversa

Negozio del barbiere



Il salone del barbiere ovviamente accetta un determinato numero di clienti, quindi inizialmente accetta un determinato numero di clienti in piedi. Alcuni di questi, sulla base di chi arriva per prima, si può sedere su un divano e tra questi alcuni possono accedere alle sedie del barbiere (si compete per entrambe le risorse). Una volta tagliati i capelli uno alla volta vanno alla cassa per pagare ed escono

Noto ciò si può supporre che ci sia una capienza massima del negozio per ciascun settore indicato da `max_clust`. Nell'esempio assegneremo un barbiere per ogni sedia

Prima soluzione

L'idea della prima soluzione, si possono servire, nel corso dell'intero periodo (es. un giorno), un numero massimo di clienti e una sola cassa per la quale competono i barbieri (ovvero, un barbiere libero a caso può fare le veci del cassiere)

```
// finish=numero massimo di persone servibili nel periodo
// max_clust=numero massimo di persone contemporaneamente nel negozio
// coord=numero di barbieri
semaphore
        max_clust=20, sofa=4, chair=3, coord=3, ready=0, leave_ch=0,
        paym=0, recpt=0, finish[50]={0};
        mutex1=1, mutex2=1;
```

```
int count = 0;

void customer() {
        int cust_nr;
        wait(max_cust);
        enter_shop();
        // essendo count una variabile condivisa la proteggo con un
semaforo
        wait(mutex1);
        cust_nr = count;
        count++;
        signal(mutex1);
        wait(sofa);
        sit_on_sofa();
        wait(chair);
        get_up_from_sofa();
        signal(sofa);
        sit_in_chair();
        // mutex2 è condivisa con il barbiere quindi metto un
semaforo
        // (barber fa dequeue e customer fa enqueue)
        wait(mutex2);
        enqueue1(cust_nr);
        // avrei potuto anche utilizzare un solo semaforo mutex ma
ciò avrebbe
        // significato una diminuzione di prestazioni
        signal(mutex2);
        signal(ready);
        // aspetta che il barbiere finisca
        wait(finish[cust_nr]);
        leave_chair;
        signal(leave_cr);
        pay();
        wait(recpt);
        exit_shop();
        signal(max_cust);
}

void barber() {
        int b_cust;
        while(true) {
                // un barbiere inizia a fare qualcosa solo quando
arriva un
                // signal(ready)
                wait(ready);
                // quale cliente servire
```

```
                wait(mutex2);
                dequeue1(b_cust);
                signal(mutex2);
                wait(coord);
                cut_hair();
                signal(coord);
                signal(finish[b_cust]);
                wait(leave_ch);
                signal(chair);
        }
}

void cashier() {
        while(true) {
                wait(paym);
                wait(coord);
                accept_pay();
                signal(coord);
                signal(recpt);
        }
}
```

Seconda soluzione

In questa seconda soluzione, non si ha limite massimo al numero di clienti servibili in un giorno e inoltre non si ha un processo separato per pagare, ma resta il fatto che si paga un barbiere qualsiasi (purché libero).
Qui inoltre viene utilizzato un solo semaforo `mutex`, non è più necessario il semaforo `coord` e ci sono tanti semafori `finish` quanti sono i barbieri. Non è più presente il semaforo `leave_ch` (che tanto era inefficiente, solo un cliente alla volta poteva alzarsi).

Il semaforo `chair` è inizializzato a 0 e viene incrementato ogni volta che un barbiere torna libero

Nonostante tutto ciò ci sta una piccola inefficienza: un solo barbiere alla volta può preparare la **propria** sedia

```
int next_barber;
void Barber(i) {
        while(true) {
                wait(mutex);
                next_barber = i;
                signal(chair);
                wait(ready);
```

```
                signal(mutex);
                cut_hair();
                signal(finish[i]);
                wait(paym);
                accept_pay();
                signal(recpt);
        }
}

void Customer(i) {
        int my_barber;
        wait(max_cust);
        enter_shop();
        wait(sofa);
        sit_on_sopfa();
        wait(chair);
        get_up_from_sofa();
        signal(sofa);
        my_barber = next_barber;
        sit_in_chair();
        signal(ready);
        wait(finish[my_barber]);
        leave_chair();
        pay();
        signal(paym);
        wait(recpt);
        exit_shop();
        signal(max_cust);
}
```