**ESERCIZI DI SINCRONIZZAZIONE TRA THREAD A LIVELLO UTENTE**

**ESERCIZIO SincrThread-1**

In un processo P sono definiti il thread MAIN, A e B , realizzati a livello utente con scheduling *round robin*  e quanto di tempo di 5 *msec.*

I thread A e B cooperano scambiando messaggi attraverso un buffer condiviso, capace di contenere un solo messaggio. Le parti rilevanti del codice dei thread A e B sono le seguenti:

Thread A:

1. **while** (**not** fine) {
2. <produce mess>
3. **lock**(&BufferVuoto**)**;
4. <deposita mess nel buffer>
5. void fun();
6. **unlock**(&BufferPieno);

}

Thread B:

1. **while** (**not** fine) {
2. **lock**(&BufferPieno);
3. <preleva mess dal buffer>
4. **unlock**(&BufferVuoto);
5. <consuma mess>

}

dove BufferPieno, BufferVuoto e fine sono variabili binarie condivise. I protocolli lock(K) e unlock(K)sono realizzati nel modo seguente:

lock(K) unlock(K)

......... ............

loop TSL K, R1 MOV K, #1

JNZ R1, SezCrit ...........

JMP loop

SezCrit ..........,

dove R1 è un registro generale del processore e K è l’indirizzo simbolico della chiave. Si conviene che le sezioni critiche siano accessibili quando la chiave ha il valore 1.

Al tempo T si ha BufferPieno= 1, BufferVuoto= 0 e fine= 0 ed è in esecuzione il thread A che esegue la riga a.1). I rimanenti thread sono pronti, con l’ordinamento B 🡪 MAIN e il thread B è pronto a eseguire la riga b.1). Il tempo di esecuzione di tutte le righe di codice di A e B è trascurabile, ad eccezione della riga a.5) che richiede 8 msec.

Si chiede il tempo necessario ad A per depositare 2 messaggi, a partire dal tempo T. Si suppone che la variabile *fine* conservi il valore 0 e che il thread MAIN consumi sempre per intero il suo quanto di tempo.

**SOLUZIONE**

# 1) Primo messaggio depositato al tempo T+ 15 . Infatti

- Thread A esegue a.1), a.2) e poi a.3) per 5 msec;

- Thread B esegue b.1), b.2), b.3), b.4), b.5), b.1), b.2) per 5 msec

- Thread MAIN utilizza il processore per 5 msec;

- Thread A esegue a.3) e a.4) per 0 msec e deposita mess

# 2) Secondo messaggio depositato al tempo T+ 15 + 30

- Thread A esegue a.5) per 5 msec);

- Thread B esegue b.2 per 5 msec;

- Thread MAIN utilizza il processore per 5 msec;

- Thread A esegue a.5) per 3 msec e quindi a.6), a.1), a.2) e poi a.3) per 2 msec

- Thread B esegue b.2), b.3), b.4), b.5, b.1), b.2) per 5 msec

- Thread MAIN utilizza il processore per 5 msec;

- Thread A esegue a.3) e a.4) e deposita mess

**ESERCIZIO SincrThread-2**

In un processo P sono definiti il thread MAIN, A e B , realizzati a livello utente con scheduling *round robin*  e quanto di tempo di 5 *msec.*

I thread A e B cooperano scambiando messaggi attraverso un buffer condiviso, capace di contenere un solo messaggio. Le parti rilevanti del codice dei thread A e B sono le seguenti:

Thread A:

1. **while** (**not** fine) {
2. <produce mess>
3. **lock**(&BufferVuoto**)**;
4. <deposita mess nel buffer>
5. void fun();
6. **unlock**(&BufferPieno);

}

Thread B:

1. **while** (**not** fine) {
2. **lock**(&BufferPieno);
3. <preleva mess dal buffer>
4. **unlock**(&BufferVuoto);
5. <consuma mess>

}

dove BufferPieno, BufferVuoto e fine sono variabili binarie condivise.

Per limitare la durata dell’attesa attiva dei thread nell’esecuzione della thread\_lock i protocolli thread\_lock e thread\_unlock utilizzano la funzione thread\_yield per rilasciare il processore. La loro realizzazione è la seguente:

thread\_lock(K) thread\_unlock(K)

......... ............

loop TSL K, R1 MOV K, #1

JNZ R1, SezCrit CALL thread\_yield

CALL thread\_yield ...........

JMP loop

SezCrit .......

dove R1 è un registro generale del processore, K è l’indirizzo simbolico della chiave e la funzione thread\_yield provoca il rilascio del processore da parte del thread che la esegue. Si conviene che le sezioni critiche siano accessibili quando la chiave ha il valore 1.

Al tempo T si ha BufferPieno= 1, BufferVuoto= 0 e fine= 0 ed è in esecuzione il thread A che esegue la riga a.1). I rimanenti thread sono pronti, con l’ordinamento B 🡪 MAIN e il thread B è pronto a eseguire la riga b.1). Il tempo di esecuzione di tutte le righe di codice di A e B è trascurabile, ad eccezione della riga a.5) che richiede 8 msec.

Si chiede il tempo necessario ad A per depositare 2 messaggi, a partire dal tempo T. Si suppone che la variabile *fine* conservi il valore 0 e che il thread MAIN consumi sempre per intero il suo quanto di tempo.

**SOLUZIONE**

# 1) Primo messaggio depositato al tempo T+ 5 Infatti:

* Thread A esegue a.1), a.2) e poi a3) per 0 msec;
* Thread B esegue b1), b2), b3), b4) per 0 msec;
* MAIN esegue il suo programma per 5 msec;
* Thread A esegue a3) e a4)per 0 msec e deposita il messaggio al tempo T+ 5;

# 2) Secondo messaggio depositato al tempo T+ 23 Infatti:

* Thread A esegue a5) (funzione fun) per 5 msec;
* Thread B esegue b5), b1) e b2) per 0 msec;
* MAIN esegue il suo programma per 5 msec;
* Thread A esegue a5) (funzione fun) per 3 msec e quindi a6) per 0 msec;
* Thread B esegue b2), b3) , b4) per 0 msec;
* MAIN esegue il suo programma per 5 msec;
* Thread A esegue a1, a2), a3), a4)per 0 msec e deposita il messaggio al tempo T= 23.

**ESERCIZIO SincrThread-3**

In un sistema con thread *realizzati a livello utente*, dove l’unità schedulabile dal kernel è il processo, sono presenti i processi P1 con thread T11, T12, T13, e P2 con thread T21 e T22. Ogni processo gestisce i suoi thread con politica Round Robin.

Immediatamente prima del tempo *t* è in esecuzione il processo P1 e il processo P2 è pronto. Nel processo P1 è in esecuzione il thread T11 e i rimanenti thread dei due processi sono pronti, con il seguente ordinamento nelle rispettive code:

processo P1: T12-> T13 (ultimo).

Processo P2: T21->T22 (ultimo).

Quale thread va in esecuzione se al tempo t si verificano (in alternativa) i seguenti eventi:

|  |  |  |
| --- | --- | --- |
|  | Evento che si verifica al tempo t | Thread in esecuzione  dopo il tempo t |
| (a) | T11 esegue una wait su un semaforo con valore 0 | T21 |
| (b) | T11 esegue una signal su un semaforo con valore 2 | T11 |
| (c) | T11 esaurisce il proprio quanto di tempo | T12 |
| (d) | Il sistema operativo scarica il processo P1 in memoria secondaria. | T21 |

**ESERCIZIO SincrThread-4**

In un sistema con thread realizzati a livello utente, sono presenti i processi P1 con thread T11, T12, T13, il processo P2 con thread T21 e T22 e il processo P3 con il solo thread T31. Per ogni processo, lo scheduler dei thread applica la politica FIFO e interviene quando un thread esegue l’operazione *thread\_yield.* La politica di scheduling dei processi non prevede il prerilascio.

Al tempo *T* è in esecuzione il processo P1, il processo P2 è pronto e il processo P3 è sospeso sul semaforo *sem1*. Nel processo P1 è in esecuzione il thread T11 e i rimanenti thread dei tre processi sono pronti, con il seguente ordinamento nelle rispettive code:

Processo P1: T12-> T13 Processo P2: T21->T22 Processo P3: T31

Si chiede qual è il contenuto della coda pronti dei processi e quale thread è in esecuzione dopo che si sono verificate (in alternativa) le due seguenti sequenze di eventi:

1. Il thread in esecuzione esegue *thread\_yield,* quindi il thread in esecuzione esegue una *signal* sul semaforo *sem1*; quindi il thread in esecuzione esegue una *wait* sul semaforo *sem2,* che ha valore 0;
2. Il thread in esecuzione esegue una *wait* sul semaforo *sem1*; il thread in esecuzione esegue una *thread\_yield*; quindi il thread in esecuzione esegue una *signal* sul semaforo *sem1.*

**SOLUZIONE**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Sequenze di eventi | Thread in esecuzione | Coda pronti |
| a-1 | Il thread in esecuzione esegue *thread\_yield* | T12 | P2 |
| a-2 | il thread in esecuzione esegue *signal*(*sem1)* | T12 | P2 🡪 P3 |
| a-3 | il thread in esecuzione esegue *wait(sem2)* con valore 0 | T21 | P3 |
|  |  |  |  |
| b-1 | il thread in esecuzione esegue *wait*(*sem1)* | T21 | - |
| b-2 | il thread in esecuzione esegue *thread\_yield* | T22 | - |
| b-3 | il thread in esecuzione esegue *signal*(*sem1)* | T22 | P3 |

**ESERCIZIO SincrThread-5**

In un sistema dove i processi operano in ambiente locale e dispongono di una libreria per la realizzazione dei thread a livello utente, sono presenti il processo P1 con priorità 10, il processo P2 con priorità 5 e il processo P3 con priorità 10. Lo scheduling dei processi avviene con una politica a priorità (va in esecuzione il processo pronto con il massimo valore di priorità) e prevede il prerilascio. Per ogni processo, i thread alternano tra lo stato di esecuzione e quello di pronto. Il thread che passa dallo stato di esecuzione a quello di pronto viene inserito nell’ultima posizione della coda dei thread pronti. Lo scheduling dei thread avviene con politica FIFO, senza prerilascio.

Al tempo *T* è in esecuzione il processo P1, il processo P2 è pronto e il processo P3 è sospeso in seguito all’esecuzione della primitiva *receive(P1, &mess)*. Inoltre sono stati creati seguenti thread:

Processo P1: thread T11, in stato di esecuzione;

Processo P2 : thread T21, T22, tutti in stato di pronto con il seguente ordinamento della coda: 🡪 T21 🡪 T22 ;

Processo P3 : thread T31, T32, T33, tutti in stato di pronto con il seguente ordinamento della coda: 🡪 T31 🡪 T32 🡪 T33.

Si chiede quale è lo stato dei tre processi e quale thread è in esecuzione **dopo** ciascuno degli eventi della sequenza riportata in tabella.

**SOLUZIONE**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SEQUENZA DI EVENTI | | DOPO L’EVENTO | | | |
| Stato di P1 | Stato di P2 | Stato di P3 | Thread in esecuzione |
| a) | il thread in esecuzione esegue la primitiva asincrona *send(P3, &messaggio)* | Esec | Pronto | Pronto | T11 |
| b) | Il thread in esecuzione esegue l’operazione *thread\_yield* | Esec | Pronto | Pronto | T11 |
| b) | il thread in esecuzione esegue la primitiva bloccante *receive(P1, &risposta)* | Sospeso | Pronto | Esec | T31 |
| d) | il thread in esecuzione commette un errore di indirizzamento non recuperabile | Sospeso | Esec | Soppresso | T21 |
| e) | il thread in esecuzione esegue la primitiva asincrona *send(&mess, P1)* | Sospeso | Esec | Soppresso | T21 |

**ESERCIZI DI SINCRONIZZAZIONE TRA THREAD A LIVELLO KERNEL**

**ESERCIZIO SincrThread-6**

In un sistema con thread *realizzati a livello kernel,* dove l’unità schedulabile è il thread, sono presenti i processi P1 con thread T11, T12, T13, e P2 con thread T21 e T22. Il processore viene gestito con politica Round Robin.

Immediatamente prima del tempo *t* è in esecuzione il thread T11 e i rimanenti thread sono pronti, con il seguente ordinamento nella coda pronti:

(primo) T12->T21->T13->T22 (ultimo).

Quale thread è in esecuzione se al tempo t si verificano (in alternativa) i seguenti eventi:

|  |  |  |
| --- | --- | --- |
|  | Evento che si verifica al tempo t | Thread in esecuzione  dopo il tempo t |
| (a) | T11 esegue una wait su un semaforo con valore 0 | T12 |
| (b) | T11 esegue una signal su un semaforo con valore 2 | T11 |
| (c) | T11 esaurisce il proprio quanto di tempo | T12 |
| (d) | Il sistema operativo scarica il processo P1 in memoria secondaria. | T21 |

**ESERCIZIO SincrThread-7**

In un sistema operativo che realizza i thread a livello kernel, il thread T1i  del processo P1 e il thread T2k del processo P2 si scambiano messaggi attraverso una pila di capacità illimitata. Si consideri la seguente soluzione realizzata con il semaforo Mutex, inizializzato ad 1.

Thread T1i

While (True) {

...............

messaggio = produce\_messaggio();

wait(&Mutex);

puntatore ++;

Pila[puntatore]= messaggio;

signal(&Mutex); {

Thread T2k

While (True) {

wait(&Mutex);

mess= Pila[puntatore];

puntatore --;

signal(&Mutex);

consuma\_messaggio(mess)

}

La soluzione è corretta? Motivare la risposta.

**SOLUZIONE**

La soluzione è errata

Motivo: thread di processi diversi non possono condividere dati

**ESERCIZIO SincrThread-8**

In un sistema operativo che realizza i thread a livello kernel, i thread T1 e T2 del processo Psi scambiano messaggi attraverso una pila a tre posizioni (la pila è indicizzata dalla variabile puntatore inizializzata a 0). Si consideri la seguente soluzione realizzata con il semaforo Mutex, inizializzato ad 1, il semaforo ElementoLibero inizializzato a 3 e il semaforo MessaggioPresente inizializzato a 0.

|  |  |
| --- | --- |
| Thread T1:  While (True) {  messaggio= produce\_messaggio()  wait(&Mutex);  wait(&ElementoLibero);  Pila[puntatore]= messaggio;  puntatore ++;  signal(&MessaggioPresente);  signal(&Mutex);  } | Thread T2:  While (True) {  wait(&Mutex);  wait(&MessaggioPresente);  puntatore --;  mess= Pila[puntatore];  signal(&ElementoLibero); signal(&Mutex);  consuma\_messaggio(mess); } |

La soluzione è corretta? Motivare la risposta.

**SOLUZIONE**

La soluzione è errata

Motivo: quando T1 trova la pila vuota (oppure quando T2 trova la pila vuota), si sospende senza rilasciare la mutua esclusione e i due thread vanno in stallo.

Nella soluzione corretta, l’operazione wait(&ElementoLibero) (oppure wait(&MessaggioPresente) deve precedere wait(&Mutex).

**ESERCIZIO SincrThread-9**

In un sistema operativo nel quale le unità schedulabili sono i threads (realizzati a livello kernel),il thread T11  del processo P1 e il thread T12 del processo P1 cooperano scambiandosi messaggi attraverso i buffer Buffer1 e Buffer 2, ciascuno di una sola cella. T11  agisce da produttore nei confronti di Buffer1 e da consumatore nei confronti di Buffer2, mentre T12  agisce da produttore nei confronti di Buffer2 e da consumatore nei confronti di Buffer1. I due buffer sono inizialmente vuoti.

Si consideri il seguente codice nella quale a Buffer1 sono associati i semafori Buffer1Vuoto (inizializzato ad 1) e Buffer1Pieno (inizializzato a 0), e a Buffer2 sono associati i semafori Buffer2Vuoto (inizializzato ad 1) e Buffer2Pieno (inizializzato a 0). Le operazioni *InsertBuffer1*, *InsertBuffer2*, *RemoveBuffer1* e *RemoveBuffer2*, sono indivisibili e pertanto eseguibili correttamente senza instaurare la mutua esclusione.

Thread T11

while (true) {

NuovoMessaggio= produce\_item();

wait(&Buffer1Vuoto);

InsertBuffer1(NuovoMessaggio);

wait (Buffer2Pieno);

Ricevuto= RemoveBuffer2()

signal(&Buffer2Vuoto)

signal(Buffer1Pieno)

consume(Ricevuto)

}

Thread T12

while (true) {

NuovoMessaggio= produce\_item();

wait(&Buffer2Vuoto);

InsertBuffer2(NuovoMessaggio);

wait (Buffer1Pieno);

Ricevuto= RemoveBuffer1()

signal(&Buffer1Vuoto)

signal(Buffer2Pieno)

consume(Ricevuto)

}

La soluzione è corretta? In caso negativo fornire una soluzione corretta

**SOLUZIONE**

a) Soluzione corretta? NO

Motivazione della risposta: le operazioni sui semafori sono eseguite in un ordine che può determinare stallo: ad esempio se entrambi i buffer sono vuoti, il thread T11 deposita su buffer1 e poi si sospende su wait (Buffer2Pieno) (la sospensione avviene prima della notifica di deposito su buffer1). Successivamente il thread T12 deposita su buffer2 ma prima di notificare il deposito si sospende a sua volta su wait (Buffer1Pieno).

b) l’interazione corretta è la seguente:

Thread T11

while (true) {

NuovoMessaggio= produce\_item();

wait(&Buffer1Vuoto);

InsertBuffer1(NuovoMessaggio);

signal(Buffer1Pieno)

wait (Buffer2Pieno);

Ricevuto= RemoveBuffer2()

signal(&Buffer2Vuoto)

consume(Ricevuto)

}

Thread T21

while (true) {

wait (Buffer1Pieno);

Ricevuto= RemoveBuffer1()

signal(&Buffer1Vuoto)

NuovoMessaggio= produce\_item();

wait(&Buffer2Vuoto);

InsertBuffer2(NuovoMessaggio);

signal(Buffer2Pieno)

consume(Ricevuto)

}

**ESERCIZIO SincrThread-10**

In un parcheggio per auto della capacità di 10 posti si accede da 2 ingressi, I1 e I2, e si esce dall’unica uscita U. Gli ingressi e le uscite sono controllati da sbarre. Le auto (indicate come A1, A2, ... An) sono thread di uno stesso processo, realizzati a livello kernel.

Per entrare nel parcheggio dall’ingresso I1 (oppure da I2), i thread si accodano a quelli già in attesa di entrare e quando ragiungono il primo posto della coda ottengono l’accesso (cioè provocano l’apertura della sbarra) se nel parcheggio esiste almeno un posto libero. Per uscire, i thread rilasciano il posto che occupavano e si sincronizzano con quelli in attesa di entrare.

Per la gestione del parcheggio si utilizzano i semafori:

* *PostiDisponibili*, inizializzato al valore 10, *(il valore esprime il numero di posti disponibili nel parcheggio)*
* *CodaI1* e *CodaI2,*  inizializzati al valore 1. *(CodaI1 contiene i thread auto in attesa di raggiungere la sbarra, analogamente CodaI2)*)
* *MutexPosto,* inizializzato a 1 *(per la selezione, l’occupazione e il rilascio di un posto nel parcheggio)*

Scrivere le procedure eseguite dai thread per entrare dall’ingresso I1 e I2 e quella per uscire dal parcheggio.

**SOLUZIONE**

***Per entrare dall’ingresso 1***

void function Ingresso1 ()

wait(&CodaI1);

wait (&PostiDisponibili);

<si alza la sbarra che lascia passare un’auto>

<si chiude la sbarra>

signal (&CodaI1);

wait (&MutexPosto);

<seleziona e occupa un posto libero>

signal (&MutexPosto)

***Per entrare dall’ingresso 2***

void function Ingresso2 ()

wait(&CodaI2);

wait (&PostiDisponibili);

<si alza la sbarra che lascia passare un’auto>

<si chiude la sbarra>

signal (&CodaI2);

wait (&MutexPosto);

<seleziona e occupa un posto libero>

signal (&MutexPosto)

***Per uscire dal parcheggio:***

void function uscita ()

wait (&MutexPosto);

<libera il posto che occupava >

signal (&MutexPosto)

<raggiunge l’uscita e provoca l’apertura della sbarra>

<esce dal parcheggio>

signal (&PostiDisponibili);

**ESERCIZIO SincrThread-11**

In un sistema operativo multithreading sono definiti i processi P1, P2, ... , Pm e i trhead TP1, TP2, ... , TPm dove TPi appartiene al processo Pi . I thread sono realizzati **a livello kernel** e costituiscono la sola unità schedulabile. I processi sono puri contenitori di risorse e non condividono dati.

I thread TP1, TP2, ... , TPm si scambiano informazione sotto forma di record di formato fisso, attraverso un vettore circolare di *N* celle, ciascuna capace di contenere un record. Ogni thread può comportarsi, a seconda delle circostanza, da produttore o da consumatore. Poiché l’ambiente non consente la condivisione di dati da parte dei thread TP1, TP2, ... , TPm, il vettore circolare è un dato privato di un ulteriore processo, denominato *BuffServer*, e i thread richiedono il deposito o l’inserzione di record utilizzando opportune primitive di comunicazione.

Le primitive di comunicazione utilizzate sono le seguenti:

* *send(destinatario, &messaggio)* che è asincrona e specifica il nome del thread destinatario;
* *send(mailbox , &messaggio)* che è asincrona e specifica il nome del mailbox al quale si vuole inviare;
* *receive(mailbox , &mitt, &mess)* che è bloccante, specifica il mailbox dal quale si vuole ricevere e restituisce il messaggio ricevuto e il nome del suo mittente.

Si utilizzano due mailbox, denominati *mailbox\_per\_deposito* e *mailbox\_per\_prelievo.* Il generico thread produttore può depositare il contenuto del propria *record* eseguendo la primitiva *send(mailbox\_per\_deposito, &record),* mentre il generico consumatore può richiedere il prelievo di un record eseguendo in sequenza le primitive *send(mailbox\_per\_prelievo, &vuoto)* e *receive(BuffServ, &mess*). Nella prima primitiva il parametro *&vuoto* non è significativo*,* mentre nella seconda il parametro *&mess* individua il buffer destinato a ricevere il record restituito da *BuffServ.*

Il processo *BuffServer* ha due thread, denominati *ThreadGestioneProduttori* e *TreadGestioneConsumatori*, che condividono il *vettore\_circolare* (della capacità di N celle e inizialmente vuoto) e i puntatori *testa* e *coda* associati al vettore circolare (entrambi inizializzati al valore 0), ed eseguono su questo buffer il deposito o il prelievo per conto dei thread produttori o consumatori.

I thread *ThreadGestioneProduttori* e *TreadGestioneConsumatori* si sincronizzano reciprocamente con i semafori *cella\_ disponibile* (valore iniziale N) e *cella\_ occupata* (valore iniziale 0), e comunicano con i thread produttori e consumatori mediante le opportune primitive di comunicazione.

Per richiedere il deposito nel vettore circolare, il generico thread che si comporta produttore esegue il seguente frammento di codice:

*…………………………*

*record = produce\_record();*

*send(mailbox\_per\_deposito, &record);*

*………………………..*

Analogamente, per richiedere il prelievo dal vettore circolare, il generico thread che si comporta da consumatore esegue il seguente frammento di codice:

*…………………………*

*send(mailbox\_per\_prelievo, &vuoto);*

*receive(BuffServ, &mess);*

*consuma\_mess(mess)*

*………………………..*

Si chiede di specificare il codice dei due thread del processo *BuffServer*.

Si chiede inoltre se è necessaria la mutua esclusione per le operazioni sui puntatori *testa* e *coda* eseguite dai due thread, motivando la risposta.

**SOLUZIONE**

ThreadGestioneProduttori

{

**while** true {

wait (cella\_ disponibile);

*//esiste almeno una cella disponibile? Se non esiste si sospende//*

receive(mailbox\_per\_deposito, &mitt, &mess);

*//riceve la richiesta di un produttore, eventualmente sospendendosi in attesa che venga inviata//*

coda= (coda+1) mod N; vettore\_circolare[coda]= mess;

*//deposista il record nel vettore circolare, gestendo il puntatore//*

signal (cella\_ occupata);

*/si sincronizza con TreadGestioneConsumatori//*

}

}

TreadGestioneConsumatori

{

**while** true {

wait (cella\_ occupata);

*//esiste almeno un record giacente? Se non esiste si sospende//*

receive(mailbox\_per\_prelievo, &mitt, &mess);

*//riceve la richiesta di un consumatore, eventualmente sospendendosi in attesa che venga inviata//*

*// non utilizza il contenuto di mess //*

send(mitt, &vettore\_circolare[testa]; testa= (testa+1) mod N;

*//invia al richiedente il primo record del buffer, quindi aggiorna il puntatore//*

signal (cella\_ disponibile);

*//si sincronizza con TreadGestioneProduttori//*

}

}

Per le operazioni sui puntatori *testa* e *coda* non è necessaria la mutua esclusione, perchè il puntatore *testa* è utilizzato solo da *TreadGestioneConsumatori*, mentre il puntatore *coda* è utilizzato solo da *TreadGestioneProduttori*.