Esercizio E1.10

Parte 1)
Impostazione

Ricordiamo che una `entry` rappresenta un servizio che il `server` offre ai processi clienti e che costituisce una coda (FIFO) di *chiamate* (richieste di servizio), dove una *chiamata* è caratterizzata da una struttura contenente i valori dei parametri di modo `in`, richiesti dalla `entry`, e il nome (il `PID`) del processo chiamante, informazione, quest'ultima, necessaria al `server` per consentirgli di restituire al chiamante i valori dei parametri di modo `out`. Inoltre, questa coda FIFO è condivisa tra i clienti (i quali vi inseriscono i valori caratterizzanti la loro *chiamata*) e il `server` che dalla coda deve estrarre i valori di una *chiamata* per svolgere il servizio richiesto. Quindi una `entry` la possiamo caratterizzare come una `mailbox` con `N` mittenti (i clienti) e un ricevente (il `server`). Nel caso specifico, avendo l'unica entry `E` un solo parametro di modo `in`, il tipo degli elementi della `mailbox` sarà costituto dai valori di una struttura con due campi: uno di tipo `T1` (tipo del parametro) ed uno di tipo `int` (il `PID` del chiamante).
Ogni processo cliente, una volta effettuata la *chiamata*, si aspetta poi di ricevere dal `server` il valore del parametro di modo `out` (risultato del servizio richiesto). Possiamo quindi supporre che, per ogni cliente sia definita una struttura condivisa (`risultato`) tra lui stesso e il `server` (in pratica una nuova mailbox di lunghezza unitaria) nella quale il `server`, alla fine del servizio, deposita il valore di tipo `T2` del parametro di modo `out` e dalla quale il cliente estrae, quando disponibile, tale valore come risultato del servizio richiesto.
Dovremo quindi definire due tipi astratti di oggetti condivisi: il tipo `mailbox` di cui l'unica istanza `E` rappresenta la `entry` e il tipo astratto *mailbox_unitaria* di cui dovremo dichiarare `N` diverse istanze, una per ognuno degli `N` processi clienti. Utilizzando tali strutture potremo poi facilmente scrivere il codice che simula sia l'istruzione `accept` che quella di `entry call`.

Soluzione

/* definizione della struttura `chamata`*/

```
typedef struct {
   T1 valore_in;
   int  cliente;
} chiamata;
```

/* definizione del tipo astratto `mailbox`*/

```
class mailbox {
   chiamata buffer[N]; /* vettore circolare. Definendo il vettore di N elementi, cioè tanti
                          quanti sono i clienti, nessuno potrà mai trovare il vettore pieno,  per
                          cui il corrispondente semaforo risorsa può essere evitato */.
   int  ultimo=0;  /* rappresenta l'indice dell'elemento di buffer in cui inserire il prossimo
                      messaggio*/
   int primo=0; /* rappresenta l'indice dell'elemento di buffer da cui il server preleva il
                   prossimo messaggio */
   semaphore mutex=1;  /*semaforo di mutua esclusione fra i vari clienti. Utilizzando un vettore
                         circolare non è necessaria la mutua esclusione tra un mittente e
                         il ricevente*/
   semaphore pieno=0;  /*semaforo risorsa che indica gli elementi pieni del buffer*/

   public void send (chiamata messaggio) {  /* funzione di invio di una chiamata*/
      P(mutex);
      buffer[ultimo]=messaggio;
```

```
      ultimo=(ultimo+1)%N;
      V(mutex);
      V(pieno);
   }

   public chiamata receive(){/* funzione di ricezione di una chiamata */
      chiamata messaggio;
      P(pieno);
      messaggio=buffer[primo];
      primo1=(primo+1)%N;
      return messaggio;
   }
}
```

/* dichiarazione della entry `E`*/

```
mailbox E;
```

/* definizione del tipo astratto `mailbox_unitaria`*/

```
class mailbox_unitaria {
   T2 buffer; /* buffer destinato a contenere il valore restituito */.
   semaphore pieno=0;
   semaphore vuoto=1;
   /*i due semafori pieno e vuoto costituiscono un semaforo binario composto*/

   public void deposita(T2 valore_out) {
      P(vuoto);
      buffer= valore_out;
      V(pieno);
   }

   public T2 estrai() {
      T2 valore_out;
      P(pieno);
      valore_out=buffer;
      V(vuoto);
      return valore_out;
   }
```


/* dichiarazione dell'array di mailbox unitarie*/

```
mailbox_unitaria risultato[N];
```

/* con tali strutture possiamo ora scrivere il codice sia dell'istruzione `accept` che di quella di `enty call`*/

/* simulazione di `accept`*/

```
   T1 a,x;
   T2 y;
   int chiamante
   chiamata ch;
   ……
   {  ch=E.receive();
      x=ch.valore_in;
      chiamante=ch.cliente;
      a=x;
      y=f(a);
      risultato[chiamante].deposita(y);
   }
```

/* simulazione di `entry call`*/

```
T1 a,;
T2 b;
chiamata ch;
……
{  ch.valore_in=a;
   ch,cliente=PIE; /*Processo In Esecuzione*/
   E.send(ch);
   y=risultato[PIE].estrai();
}
```

Parte 2a)

Impostazione

Se vi è un solo cliente, anche la `mailbox` che costituisce il tipo della `entry E` viene ad essere una mailbox unitaria. Per cui è sufficiente definire solo il tipo astratto `mailbox_unitaria` e di questa dichiarare due sole istanze: l'istanza `E` che rappresenta la `entry` e l'istanza `risultato`. Con tale semplificazione le due simulazioni diventano:

Soluzione

/* simulazione di `accept`*/

```
T1 a,x;
T2 y;
int chiamante
chiamata ch;
……
{  ch=E.estrai();
   x=ch.valore_in;
   chiamante=ch.cliente;
   a=x;
   y=f(a);
   risultato.deposita(y);
}
```

/* simulazione di `entry call`*/

```
T1 a,;
T2 b;
chiamata ch;
……
{  ch.valore_in=a;
   ch.cliente=PIE; /*Processo In Esecuzione*/
   E.deposita(ch);
   y=risultato.estrai();
}
```


Parte 2b)

Impostazione

Se la entry `E` ha un solo parametro di modo `in` significa che il `server` non deve restituire nessun risultato. Ma il cliente deve però rimanere sincronizzato (sospeso) fino alla fine del servizio da parte del `server`. Ciò significa che il `server` deve comunque restituire almeno un segnale. Quindi è sufficiente sostituire all'array `risultato` un array di *semafori evento*, uno per ciascun cliente e non è perciò necessario definire il tipo astratto mailbox unitaria.

Soluzione

/* definizione della struttura `chamata`*/

```
typedef struct {
   T1 valore_in;
```

```
   int  cliente;
} chiamata;
```

/* definizione del tipo astratto `mailbox`*/

```
class mailbox {
   chiamata buffer[N]; /* vettore circolare. Definendo il vettore di N elementi, cioè tanti
                          quanti sono i clienti, nessuno potrà mai trovare il vettore pieno, per
                          cui il corrispondente semaforo risorsa può essere evitato */.
   int ultimo=0;  /* rappresenta l'indice dell'elemento di buffer in cui inserire il prossimo
                     messaggio*/
   int primo=0; /* rappresenta l'indice dell'elemento di buffer da cui il server preleva il
                   prossimo messaggio */
   semaphore mutex=1; /*semaforo di mutua esclusione fra i vari clienti. Utilizzando un vettore
                        circolare non è necessaria la mutua esclusione tra un mittente e
                        il ricevente*/
   semaphore pieno=0; /*semaforo risorsa che indica gli elementi pieni del buffer*/

   public void send (chiamata messaggio) {  /* funzione di invio di una chiamata*/
      P(mutex);
      buffer[ultimo]=messaggio;
      ultimo=(ultimo+1)%N;
      V(mutex);
      V(pieno);
   }

   public chiamata receive(){/* funzione di ricezione di una chiamata */
      chiamata messaggio;
      P(pieno);
      messaggio=buffer[primo];
      primo1=(primo+1)%N;
      return messaggio;
   }
}
```


/* dichiarazione della entry `E`*/

```
mailbox E;
```

/* dichiarazione dell'array di *semafori evento*/

```
semaphore risultato[N]={0, ……, 0};
```

/* con tali strutture possiamo ora scrivere il codice sia dell'istruzione `accept` che di quella di `enty call`*/

/* simulazione di `accept`*/

```
   T1 a,x;
   int chiamante
   chiamata ch;
   ……
   {  ch=E.receive();
      x=ch.valore_in;
      chiamante=ch.cliente;
      <corpo dell'accept>;
      V(risultato[chiamante]);
   }
```

/* simulazione di `entry call`*/

```
   T1 a,;
```

```
chiamata ch;
……..
{  ch.valore_in=a;
   ch,cliente=PIE; /*Processo In Esecuzione*/
   E.send(ch);
   P(risultato[PIE]);
}
```

Parte 2c)

Impostazione

Se la `entry E` non ha parametri la struttura `chiamata` coincide con l'intero che rappresenta il nome del `cliente`. Inoltre, se `accept` non ha corpo, una chiamata e il corrispondente `accept` costituiscono un semplice meccanismo di sincronizzazione. Quindi, come nel caso precedente, la mailbox unitaria si riduce ad un semplice *semaforo evento* su cui si sopende il chiamante in attesa che il `server` accetti la sua chiamata:

Soluzione

/* definizione del tipo astratto `mailbox`*/

```
class mailbox {
   int buffer[N];
   int ultimo=0;
   int primo=0;
   semaphore mutex=1;
   semaphore pieno=0;

   public void send (int cliente) {
      P(mutex);
      buffer[ultimo]=cliente;
      ultimo=(ultimo+1)%N;
      V(mutex);
      V(pieno);
   }

   public int receive(){/* funzione di ricezione di una chiamata */
      int chiamante;
      P(pieno);
      chiamante=buffer[primo];
      primo1=(primo+1)%N;
      return chiamante;
   }
}
```

/* dichiarazione della entry `E`*/

```
mailbox E;
```

/* dichiarazione dell'array di *semafori evento*/

```
semaphore risultato[N]={0, ……, 0};
```

/* con tali strutture possiamo ora scrivere il codice sia dell'istruzione `accept` che di quella di `enty call`*/

/* simulazione di `accept`*/

```
   int chiamante
   ……..
   {  chiamante= E.receive();
      V(risultato[chiamante]);
```

```
    }
/* simulazione di entry call*/
    ……
    {   E.send(PIE);
        P(risultato[PIE]);
    }
```