Esercizio E1.9

Parte prima)
Impostazione

Per garantire la mutua esclusione fra le funzioni del monitor è necessario riservare un semaforo (`mutex`) inizializzato a uno. Inoltre, per ogni variabile (`cond`) di tipo `condition` è necessario riservare un *semaforo condizione* (`condsem`) e un associato intero (`condcont`) per tener contro dei processi sospesi sulla condizione stessa.

Soluzione

/* struttura dati che il compilatore associa ad ogni stanza del monitor*/

```
semaphore mutex=1; /* semaforo di mutua esclusione fra le funzioni public del monitor*/
/* per ogni variabile di tipo condition presente nelle struttura dati del monitor il compilatore riserva due
   variabili: un semaforo condizione e un intero. Ad esempio se cond è una variabile di tipo condition del
   monitor, il compilatore riserva le seguenti variabili: */
semaphore condsem=0;
int condcount=0;
```

/* per ogni funzione `public` del monitor il compilatore espande, rispettivamente in testa e in coda, i seguenti `prologo` ed `epilogo`: */

```
prologo =>  P(mutex);
epilogo =>  V(mutex);
```


/* infine, il compilatore traduce nel modo seguente le operazioni sulle variabili `condition`: */

```
wait(cond) =>          { condcount++;
                         V(mutex);
                         P(condsem);;
                         P(mutex);
                       }

signal(cond) =>        { if(condcount!=0) {;
                             condcount--;
                             V(condsem);
                          }
                       }

signalAll(cond) =>     { if(condcount!=0)
                             while(condsem!=0) {
                                condcount--;
                                V(condsem);
                             }
                       }
```

Parte seconda)
Impostazione

La semantica *signal_and_urgent_wait* presuppone che il processo svegliato possa riprendere immediatamente l'esecuzione dentro il monitor mentre il processo segnalante si dovrà sospendere

cedendogli il passo (tipicamente ciò si ottiene mediante la tecnica del *passaggio del testimone*). In questo caso, però, il processo segnalante non può svegliare più di un processo alla volta altrimenti i processi svegliati riprenderebbero tutti contemporaneamente la loro esecuzione in sezione critica.
Anche utilizzando la semantica *signal_and_urgent_wait* possiamo, comunque, svegliare tutti i processi sospesi su una variabile `cond` di tipo `condition` con la seguente istruzione che ha lo stesso effetto di una `signalAll`:

```
while(!empty(cond)) signal(cond);
```

anche se l'effetto viene ottenuto con un livello di efficienza molti più basso. Infatti, supponiamo che su `cond` siano sospesi 10 processi. L'istruzione precedente implica eseguire, in questo caso, il ciclo `while` per 10 volte e, durante ogni ciclo il processo in esecuzione sveglia uno dei processi sospesi e gli cede il passo sospendendosi mediante un cambio di contesto. Quando il processo svegliato esce dal monitor, il processo segnalante riprende la sua esecuzione (nuovo cambio di contesto) e svegliando il secondo processo sospeso ripete il passo precedente mediante un ulteriore cambio di contesto e così via per ognuno dei 10 cicli.

Una soluzione più efficiente per ottenere lo stesso effetto di una `signalAll(cond)` con la semantica *signal_and_urgent_wait* ma svegliando sempre un solo processo alla volta, è quella di obbligare l'unico processo svegiato, quando lo stesso sta per uscire dal monitor, a verificare se ci siano, sulla stessa variabile `cond` altri processi che possano essere svegliati e, se ce ne sono svegliarne un altro e così via. Ad esempio se nella funzione `op1` del monitor un processo si sospende su `cond` se la condizione `B` è falsa e se nella funzione `op2`, verificato che `B` è vera, supponiamo che si possano svegliare tutti i processi sospesi su `cond`, se ne sveglia comunque uno solo e, alla fine di `op1` il processo svegliato eventualmente sveglia il secondo e così via:

```
public void op1() {
    if(!B) wait(cond);
    ……
    ……
    signal(cond);
}

public void op2 {
    ……
    ……
    /*supponendo che qui B sia versa e che si possano svegliare tutti i processi sospesi su cond*/
    signal(cond);
}
```


Con questo criterio, se N sono i processi sospesi che alla fine di `op2` possono essere svegliati, quando tutti sono stati riattivati e il processo segnalante (quello che esegue la `op2`) riprende la sua esecuzione, si saranno verificati N+1 cambi di contesto; N per riattivare, uno alla volta, gli N processi sospesi con la tecnica del *passaggio del testimone* e l'ultimo per ricedere il controllo al processo segnalante.

Viceversa, con l'istruzione `while(!empty(cond)) signal(cond)` che simula una `signalAll` si hanno 2*N cambi di contesto (2 per ogni processo risvegiato: uno per sveglare quel processo e uno per cedere di nuovo il controllo al processo segnalante).