COMPITO CON SOLUZIONI

# Sistemi Operativi (modulo II / B)

9 settembre 2016

## Esercizio 1

Considerare N thread `cliente` che attendono di essere serviti da uno degli M thread `servente`. Il thread `cliente` prende un numero univoco salvandosi localmente il valore della variabile condivisa `numero` e incrementandola, annuncia di avere tale numero e poi attende che il proprio numero venga servito. Il thread `servente` si salva localmente il numero del cliente da servire (variabile condivisa `num_da_servire`, che incrementa), attende che ci sia un `cliente` con tale numero e poi lo serve.

```
int numero = 0; num_da_servire = 0; // variabili condivise

thread cliente {
  int n; // locale
  P(mutex_cliente);
  n = numero;
  numero++;
  V(mutex_cliente);
  // annuncia di avere il numero n
  V(in_attesa[n]);
  // attende di essere servito
  P(servito[n]);
}
```

```
thread servente {
  int s; // locale
  P(mutex_servente);
  s = num_da_servire;
  num_da_servire++;
  V(mutex_servente);
  // attende che ci sia il cliente con numero
s
  P(in_attesa[s]);
  // serve (sblocca) il cliente con numero s
  V(servito[s]);
}
```

Aggiungere le operazioni di P e V sui semafori, indicandone i valori di inizializzazione. Fare attenzione alle interferenze sulle variabili condivise. Spiegare brevemente la soluzione proposta:
Utilizziamo i semafori `mutex_cliente` e `mutex_servente`, inizializzati a 1 (verde), per gestire, rispettivamente, la mutua esclusione sulle variabili `numero` e `num_da_servire`. In questo modo evitiamo le interferenze sulle variabili globali che potrebbero assegnare lo stesso numero a due thread.
Utilizziamo poi due array di semafori `in_attesa[N]` e `servito[N]`, inizializzati a 0 (rosso), per sincronizzare i clienti e i serventi. Il cliente annuncia di essere in attesa eseguendo `V(in_attesa[n])` mentre il servente attende il cliente eseguendo `P(in_attesa[s])`. In modo analogo, il servente serve il cliente eseguendo `V(servito[s])` e il cliente attende di essere servito con `P(servito[n])`.

## Esercizio 2

Considerare il monitor `risorse` che alloca le risorse `file[0]` e `file[1]` ai thread che ne fanno richiesta:

```
Monitor risorse {
  file[2] = {true, true};

  void richiedi(int i) {
    while (!file[i]) wait();
    file[i] = false;
  }
}
```

```
void rilascia (int i) {
  file[i] = true;
  notify();
}
```

Discutere la possibilità di stallo nel caso due thread invochino, rispettivamente, `risorse.richiedi(0); risorse.richiedi(1)` e `risorse.richiedi(1); risorse.richiedi(0)` e proporre due diverse soluzioni a tale problema.
Se `risorse.richiedi(0)` e `risorse.richiedi(1)` vengono eseguite rispettivamente dai due thread ci troviamo in una situazioni di stallo in quanto le successive richieste saranno entrambi bloccanti, in modo analogo a quanto accade nel problema dei filosofi a cena (situazione di attesa circolare).

Soluzione 1: una possibile soluzione è ammettere la richiesta incrementale di risorse solo secondo un ordine prestabilito (allocazione gerarchica). Ad esempio possiamo stabilire che `file[0]` < `file[1]` e forzare il secondo thread a chiedere le risorse in tale ordine, ritornando un errore in caso contrario.

Soluzione 2: una seconda soluzione è evitare l'allocazione incrementale scrivendo un metodo che allochi entrambi i file, se disponibile, al primo thread che ne faccia richiesta. Allocando le risorse in modo "atomico" si previene lo stallo ma, solitamente, si perde l'ordine d'arrivo causando, in alcuni casi, starvation.