

Corso di Sistemi Operativi e Reti

Prova scritta 9 FEBBRAIO 2021

ISTRUZIONI PER CHI SI TROVA ONLINE:

1. **Questo file contiene il testo che ti è stato dato ieri, incluso il codice;**
2. **Mantieni a tutto schermo** questo file per tutta la durata della prova; puoi scorrere liberamente tra le sue pagine, ma non puoi cambiare applicazione;
3. **Firma** preliminarmente il foglio che userai per la consegna con nome cognome e matricola;
4. **Svolgi** il compito; puoi usare solo carta, penna e il tuo cervello;
5. **Alla scadenza** termina *immediatamente* di scrivere, e attendi di essere chiamato, pena l'esclusione dalla prova;
6. **Quando è il tuo turno** mostra il foglio ben visibile in webcam, e poi metti una foto dello stesso foglio in una chat privata Microsoft Teams con il prof.

ESERCIZIO 1, TURNO 2 - PROGRAMMAZIONE MULTITHREADED

Si supponga che i consumatori di elementi di un `RunningSushiBuffer` (e cioè i Clienti) consumino gli elementi in gruppi e che possano escludere una certa tipologia di elemento da prelevare. Si aggiunga dunque al codice pre-esistente il metodo

```
getList(self, N, t, i)
```

dove `N` è un valore intero positivo. Il metodo estrae consecutivamente `N` elementi via via che questi sono disponibili in posizione `i`, ma solo se diversi da `t`. Viene infine restituita una lista composta dagli elementi estratti.

Tutti gli elementi `t'` tali per cui `t' == t` non devono essere invece prelevati e dunque lasciati scorrere. Si noti che è vietato invocare esplicitamente `shift` dall'interno di questo metodo.

Si effettuino tutte le ulteriori modifiche che si ritengono necessarie nel resto del codice.

MATERIALE PER LA PROVA SULLA PROGRAMMAZIONE MULTI-THREADED

Il codice fornito implementa un `RunningSushiBuffer`. Tale struttura dati è costituita da un particolare tipo di buffer circolare thread-safe con N slot in cui è possibile inserire degli elementi solo in posizione 0. L'operazione di estrazione può essere invece eseguita da tutte le altre posizioni (da 1 a $N-1$). Inoltre, gli elementi del `RunningSushiBuffer` possono essere arbitrariamente ruotati di X posizioni in senso orario (Si veda la figura 1).

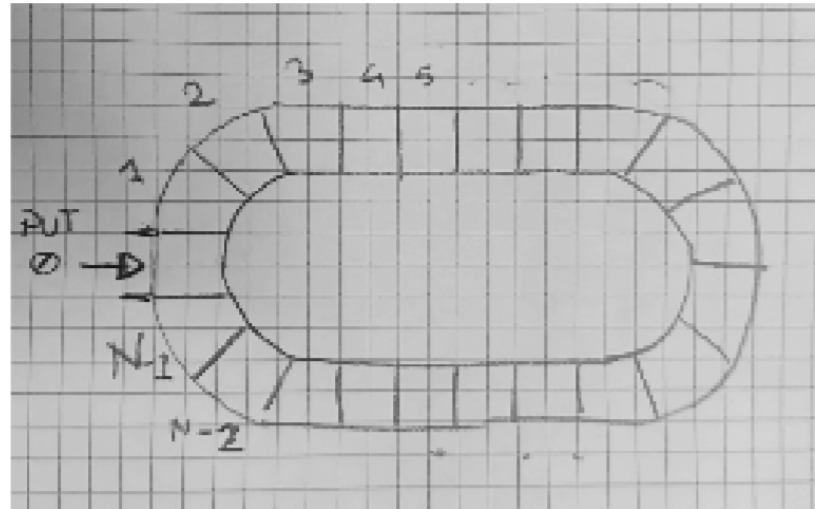
I metodi che sono stati implementati sono i seguenti:

`put(self, t)`: Si pone in attesa bloccante fintantoché la posizione 0 è occupata. Una volta usciti dall'attesa bloccante, inserisce l'elemento t in posizione 0.

`get(self, i)`: Si pone in attesa bloccante se la posizione i è libera. Rimuove e restituisce l'elemento in posizione i quando questo si rende disponibile.

`shift(self, j)`: Ruota di j posizioni il buffer circolare in senso antiorario. La rotazione è da intendersi come una rinumerazione degli indici dove ogni elemento accessibile in generica posizione X viene passato in posizione $(X+j) \bmod N$, dove N è la dimensione del buffer. Ad esempio se $j=1$, l'elemento in posizione 0 deve diventare quello che prima era in posizione 1, l'elemento 1 diventa quello che prima era in posizione 2, e così via. Essendo il buffer circolare, l'elemento che prima era in posizione 0 deve trovarsi in posizione $N-1$.

Si noti che, per evitare inutili operazioni di copia, lo spostamento è stato ottenuto con un meccanismo di rimappatura dinamica degli indici.



Nel codice vengono forniti a corredo anche dei Thread di esempio delle seguenti tipologie:

1. **Cuoco**. Un cuoco produce periodicamente degli elementi che vengono depositati in posizione 0;
2. **Cliente**. Un cliente viene posizionato su una specifica posizione p che non cambia mai, ed estrae periodicamente degli elementi da questa posizione
3. **NastroRotante**. Il nastro rotante muove periodicamente le posizioni del buffer stesso.

```

1      #!/usr/bin/env python
2
3from threading import Lock,RLock, Condition, Thread
4from time import sleep
5from random import random, randint
6
7debug = True
8
9#
10# Stampa sincronizzata
11#
12plock = Lock()
13def sprint(s):
14     with plock:
15         print(s)
16#
17# Stampa solo in debug mode
18#
19def dprint(s):
20     with plock:
21         if debug:
22             print(s)
23
24class RunningSushiBuffer:
25
26     theBuffer : list
27     dim : int
28     lock : RLock
29     condition : Condition
30
31     def __init__(self, dim):
32         self.theBuffer = [None] * dim
33         self.zeroPosition = 0
34         self.dim = dim
35         self.lock = RLock()
36         self.condition = Condition(self.lock)
37
38     def _getRealPosition(self,i : int):
39         return (i + self.zeroPosition) % self.dim

```

```

40
41 def get(self, pos : int):
42     with self.lock:
43         while self.theBuffer[self._getRealPosition(pos)] == None:
44             self.condition.wait()
45         palluzza = self.theBuffer[self._getRealPosition(pos)]
46         self.theBuffer[self._getRealPosition(pos)] = None
47         return palluzza
48
49 def put(self, t):
50     with self.lock:
51         while self.theBuffer[self._getRealPosition(0)] != None:
52             self.condition.wait()
53         self.theBuffer[self._getRealPosition(0)] = t
54
55 def shift(self, j = 1):
56     with self.lock:
57         #
58         #  uso zeroPosition per spostare la posizione 0 solo virtualmente,
59         #  anziche' dover ricopiare degli elementi
60         #
61         self.zeroPosition = (self.zeroPosition + j) % self.dim
62         #
63         #  E' solo grazie a uno shift che puo' crearsi la condizione per svegliare un thread
64         #  in attesa, rispettivamente su put() o su get()
65         #
66         self.condition.notifyAll()
67
68 class NastroRotante(Thread):
69
70     def __init__(self, d : RunningSushiBuffer):
71         super().__init__()
72         self.iterazioni = 10000
73         self.d = d
74
75     def run(self):
76         while(self.iterazioni > 0):
77             sleep(0.1)
78             self.iterazioni -= 1
79             self.d.shift()

```

```

80
81class Cuoco(Thread):
82
83    piatti = [ "*", ";", "^", "%" ]
84
85    def __init__(self, d : RunningSushiBuffer):
86        super().__init__()
87        self.iterazioni = 1000
88        self.d = d
89
90    def run(self):
91        while(self.iterazioni > 0):
92            sleep(0.5 * random())
93            self.iterazioni -= 1
94            randPiatto = randint(0, len(self.piatti)-1)
95            self.d.put(self.piatti[randPiatto])
96            print ( f"Il cuoco {self.ident} ha cucinato <{self.piatti[randPiatto]}>")
97            print ( f"Il cuoco {self.ident} ha finito il suo turno e va via")
98
99class Cliente(Thread):
100
101    def __init__(self, d : RunningSushiBuffer, pos : int):
102        super().__init__()
103        self.coseCheVoglioMangiare = randint(1,20)
104        self.d = d
105        self.pos = pos
106
107    def run(self):
108        while(self.coseCheVoglioMangiare > 0):
109            sleep(5 * random())
110            self.coseCheVoglioMangiare -= 1
111            print ( f"Il cliente {self.ident} aspetta cibo")
112            print ( f"Il cliente {self.ident} mangia <{self.d.get(self.pos)}>")
113            print ( f"Il cliente {self.ident} ha la pancia piena e va via")
114
115
116size = 20
117D = RunningSushiBuffer(size)
118NastroRotante(D).start()
119for i in range(0,2):

```

```
120     Cuoco(D).start()  
121 for i in range(1,size):  
122     Cliente(D,i).start()
```


ESERCIZIO 2, TURNO 2 - PERL

Il file `dump.log` contiene alcune informazioni riguardanti le connessioni di rete in entrata e uscita.

Una riga di esempio del file è così composta:

```
TIMESTAMP IP IP_SORGENTE.PORTA_SORGENTE > IP_DESTINAZIONE.PORTA_DESTINAZIONE: PROTOCOLLO, altri_parametri
```

Esempio Reale

```
14:25:51.932550 IP 160.97.62.90.62536 > 224.0.0.252.5355: UDP, length 21
14:26:10.171155 IP 23.6.123.119.443 > 192.168.0.100.44664: Flags [.] , ack 1, win 990, options [nop,nop,TS val 254276428 ecr 3274039351], length 0
```

Lo scopo dell'esercizio è quello di creare uno script Perl dal nome `dump.pl` che leggerà dagli argomenti a linea di comando il nome del file di log (in questo caso `dump.log`), un indirizzo ip **destinazione** (IP), una porta **destinazione** (P) e un intero (D).

Lo script dovrà essere quindi richiamato nel seguente modo:

```
./dump.pl dump.log 160.97.62.90 443
```

dove:

```
Nome File = dump.log
IP        = 160.97.62.90
P         = 443
```

Lo script dovrà essere in grado di filtrare e contare tutte le connessioni avvenute tra un qualsiasi indirizzo ip sorgente e porta sorgente verso l'indirizzo ip destinazione (specificato dal parametro **IP**) e porta destinazione (specificata dal parametro **P**).

Per ogni indirizzo ip **sorgente** che verifica la condizione di cui sopra, bisognerà contare il numero di connessioni avvenute; infine, lo script creerà un file di log dal nome `output.log` in cui sarà riportato l'elenco di tutti gli indirizzi ip sorgente che hanno effettuato una connessione all'indirizzo **ip.porta** destinazione specificati ordinati per numero di connessioni in ordine decrescente.

Esempio:

Nel caso in cui lo script sia invocato con i parametri **IP = 216.58.205.200** e **P = 443**, bisognerà prendere in considerazione soltanto le righe il cui `ip_destinazione` è **216.58.205.200** e la cui `porta_destinazione` è **443**, quindi:

Nel caso in cui il file `dump.log` contenga solo le 4 linee riportate qui di seguito, lo script filtrerà solo le ultime 2.

```
14:25:53.325453 IP 160.97.62.90.47172 > 216.58.205.200.443: UDP, length 1350 → OK (L'indirizzo ip destinazione e la porta sono corretti)
15:34:07.952457 160.97.62.90.443 > 192.168.0.100.443 Flags [P.], seq 165408:165546, ack 20306, win 425, options [nop,nop,TS val 3121993885 ecr 78163997], length 138 --> NO (La porta corrisponde ma l'indirizzo ip destinazione è diverso da quello cercato)
15:34:08.136165 IP 160.97.62.90.49736 > 216.58.205.200.443: UDP, length 41 --> OK (L'indirizzo ip destinazione e la porta sono corretti)
15:55:01.124565 IP 192.168.1.107.49236 > 216.58.205.200.443: UDP, length 41 --> OK (L'indirizzo ip destinazione e la porta sono corretti)
```

Esempio del formato file `output.log` (Nota l'ordine di stampa decrescente):

```
160.97.62.90 > 216.58.205.200.443    --->    2
192.168.1.107 > 216.58.205.200.443  --->    1
```

PROGRAMMAZIONE IN PERL - MATERIALE PRELIMINARE

Il file `dump.log` contiene alcune informazioni riguardanti le connessioni di rete in entrata e uscita.

Una riga di esempio del file è così composta:

```
TIMESTAMP IP IP_SORGENTE.PORTA_SORGENTE > IP_DESTINAZIONE.PORTA_DESTINAZIONE: PROTOCOLLO, altri_parametri
```

Esempio Reale

```
14:25:51.932550 IP 160.97.62.90.62536 > 224.0.0.252.5355: UDP, length 21
14:26:10.171155 IP 23.6.123.119.443 > 192.168.0.100.44664: Flags [.] , ack 1, win 990, options [nop,nop,TS val
254276428 ecr 3274039351], length 0
14:25:53.114205 IP 192.168.0.1.50080 > 239.255.255.250.1900: UDP, length 273
15:25:53.218232 IP 192.168.0.1.50080 > 239.255.255.250.1900: UDP, length 336
14:22:53.322898 IP 192.168.0.1.62536 > 192.168.0.100.32865: UDP, length 21
14:25:53.322916 IP 192.168.0.100 > 192.168.0.1: ICMP 192.168.0.100 udp port 32865 unreachable, length 148
14:27:53.323570 IP 192.168.0.1.53 > 192.168.0.100.48932: ICMP 192.168.0.100 udp port 32865 unreachable, length 148
16:25:53.323589 IP 192.168.0.100 > 192.168.0.1: ICMP 192.168.0.100 udp port 48932 unreachable, length 130
```