# Esercitazione 4

## Parte prima: pattern per producer / consumer

Per iniziare a prendere confidenza con l'uso dei thread e dei pattern di sincronizzazione associati, consideriamo un caso tipico legato all'esecuzione concorrente: il modello produttori e consumatori. In questo modello, un certo numero di attività (i produttori) generano informazioni che devono essere ulteriormente elaborate da altre attività (i consumatori). Il numero di produttori è indipendente da quello dei consumatori (possono essere in rapporto 1:1, 1:N, N:1, N:M). Il sistema deve rispettare alcuni vincoli fondamentali: ogni produttore può generare un numero arbitrario (ma finito) di informazioni; le informazioni generate da un singolo produttore possono essere elaborate da qualsiasi consumatore; ogni informazione prodotta deve essere elaborata esattamente una volta sola (non deve capitare, cioè, che due consumatori elaborino la stessa informazione, né che un singolo consumatore elabori un'informazione due volte, né che un'informazione non sia elaborata da nessun consumatore); i produttori possono richiedere un tempo arbitrario per generare le proprie informazioni; quando tutti i produttori hanno terminato il proprio lavoro, i consumatori finire di elaborare gli elementi  presenti nel sistema e poi terminare ordinatamente.
Questo tipo di problema trova una naturale implementazione in un sistema dove produttori e consumatori sono costituiti da thread che si appoggiano ad una struttura condivisa (thread-safe) per gestire il passaggio delle informazioni.

Definiamo una classe Jobs che utilizzeremo per passare dei task tra dei producer e dei consumer.

```
template <class T>
class Jobs {
public:
	// inserisce un job in coda in attesa di essere processato, può
	// essere bloccante se la coda dei job è piena
	void put(T job);

	// legge un job dalla coda e lo rimuove
	// si blocca se non ci sono valori in coda
	T get();

};
```

1. Implementare con un single producer e multiple consumer un sistema che ricerca una data stringa in tutti i file di testo contenuti in una directory. Tale sistema è composto da:
    a. Un'istanza della classe Jobs (detta jobs)
    b. un main che riceve come parametri: la directory in cui sono contenuti i file (solo file con estensione .txt) e una regular expression che specifica il pattern da cercare.
    c. un *producer* che scandisce la directory e legge in sequenza i file riga per riga, inserendole in jobs.
    d. n *consumer* che leggono le righe da jobs, cercano se la regex è presente per ciascuna riga e ne stampano ogni occorrenza: nome del file, numero riga e match (attenzione alla sincronizzazione dell'output)

2. sincronizzare producer e consumer, sospendendo i consumer se non vi sono dati disponibili, inserendo in put() e get() le primitive di sincronizzazione necessarie
    a. cosa capita quando i consumer sono troppo lenti rispetto alla capacità di produrre del consumer? Modificare la put(), rendendola bloccante nel caso in cui la coda dei job superi una dimensione fissata
    Quante condition variable servono in questo caso?
    b. fare attenzione a gestire in modo corretto le condizioni di terminazione, garantendo che tutti i consumer terminino senza perdersi dei job quando il producer ha finito, modificando l'interfaccia del metodo get() e della classe Jobs se necessario; provare differenti strategie, discutendone pro e contro:
        i. inviare un valore sentinella che indica la terminazione
        ii. una variabile che indica se ancora writer è attivo o meno; come può essere implementata questa variabile?
3. Implementare multiple producer e multiple consumer
    a. replicare tutti i punti del passo precedente, dividendo su più producer il ruolo di lettura dei file
    b. il main thread legge i nomi dei file e li inserisce in una coda di Jobs chiamata fileJobs, m consumer leggono i file e producono ciascuno una serie di linee in una seconda coda di Jobs detta lineJobs, da cui altri consumer che eseguono la ricerca leggono le righe e cercano se vi sono match con la regex
    c. anche in questo caso provare le differenti strategie di corretta terminazione
4. Implementare la coda dei job come un buffer circolare utilizzando un std::vector di dimensione fissa (https://en.wikipedia.org/wiki/Circular_buffer)
    a. quali vantaggi può avere rispetto ad una coda realizzata come una lista?

# Seconda parte: map / reduce utilizzando i thread

Riprendere l’esercizio della esercitazione 3 e implementarlo mediante un pool di thread che esegue la map e un reducer

1. Il main thread legge le linee del file di log e le inserisce in una coda sincronizzata lineJobs
2. I mapper consumano le linee da lineJobs, le processano ed inseriscono i result in una seconda coda resultJobs
3. Il reducer consuma da resultJobs e scrive i risultati nella struttura di accumulazione
4. gestire in modo corretto le condizioni di terminazione: il programma deve stampare tutti i risultati quando l’ultimo resul è stato inserito nella struttura di accumulazione
5. Provare a pensare ad una soluzione con più reducer in parallelo. E’ possibile? Se sì quali accorgimenti occorre prendere? Provare ad implementare una soluzione con più reducer.