# Programmazione di Sistema
# **Lab 4 -** Processi

## Obbiettivi

- pattern per la parallelizzazione di task
- utilizzare primitive per la gestione di processi (fork)
- comunicazione tra processi tramite pipe
- strategie di serializzazione dei dati
- evitare ritardi in lettura/scrittura mediante IO asincrono e select

## Premessa

Questo laboratorio utilizza le primitive `fork()` e `pipe()` dei sistemi operativi Unix-like. Pertanto è necessario servirsi di un ambiente di esecuzione adeguato (una virtual machine con Linux, ad esempio). A meno di utilizzare system call particolari, il codice dovrebbe funzionare senza grossi problemi all'interno di Mac OSX.

## Introduzione

Il paradigma *MapReduce* è un pattern utilizzato per processare grandi dataset quando il tipo di analisi da effettuare è parallelizzabile. Il sistema è composto da tre tipi di attori:

- un coordinatore
- un pool di processi *mapper*, che ricevono dal coordinatore i dati su cui lavorare (task di lavoro) in parallelo e restituiscono al coordinatore una lista di risultati, ognuno associato ad una chiave:
  `task -> [(key_1, result_1),..., (key_n,result_n)]`
- un pool di processi *reducer*, che ricevono dal coordinatore i risultati prodotti dai mapper, assieme a un accumulatore che tiene traccia dei risultati precedentemente ottenuti dai reducer associati alla stessa chiave, e restituiscono un accumulatore aggiornato con il valore ricevuto:
  `(key, result, acc)-> (key, acc)`

Questo pattern, ad esempio, può essere usato per contare la frequenza di simboli in un flusso di dati. Si consideri ad esempio questa sequenza di dati:

```
aaa bbb aaa ccc bbb aaa aaa
ccc bbb bbb aaa ccc bbb ddd
```

Il coordinatore leggerà una riga per volta e la passerà ad uno dei mapper, il quale restituirà una lista di risultati formati da coppie ordinate `(simbolo, occorrenze)`. Quindi come risultato avremo:

```
aaa bbb aaa ccc bbb aaa aaa ->[ (aaa, 4) (bbb, 2) (ccc, 1) ]
ccc bbb bbb aaa ccc bbb ddd -> [ (aaa, 1) (bbb, 3) (ccc, 1) (ddd, 1)]
```

Con due o più mapper l'elaborazione delle linee può essere fatta in parallelo distribuendo, a turno, ogni nuovo task ad un mapper differente. Man mano che il coordinatore riceve i risultati, li passa ai reducer, assieme all'accumulatore corrispondente (che in questo caso esegue la somma, partendo dal valore iniziale zero). Continuando con i dati dell'esempio:

```
(aaa, 4, 0) -> (aaa, 4)
(bbb, 2, 0) -> (bbb, 2)
(ccc, 1, 0) -> (ccc, 1)
(aaa, 1, 4) -> (aaa, 5)
(bbb, 3, 2) -> (bbb, 5)
(ccc, 2, 1) -> (ccc, 3)
(ddd, 1, 0) -> (ddd, 1)
```

Leggendo il valore degli accumulatori abbiamo in tempo reale il risultato per ogni chiave.

## Esercizio 1

Realizzare un sistema semplificato di *MapReduce* costituito da un coordinatore e due processi figli, un mapper e un reducer, che comunicano con il coordinatore mediante l'utilizzo di pipe. L'input da processare è un file di log da leggere per righe, da costruire secondo il seguente formato:

**Formato del file di log**

```
<indirizzo ip> - - [<data ora>] "<metodo> <url> <protocollo>" <codice risultato> <dimensione risultato>
```

**Esempio**:

```
209.17.96.34 - - [01/Jan/2020:00:19:59 +0100] "GET / HTTP/1.1" 200 20744
194.55.187.131 - - [01/Jan/2020:00:44:56 +0100] "GET /manager HTTP/1.1" 404 1999
```

Le funzioni map e reduce sono definite nel seguente modo generico:

```
template<class MapperInputT, class ResultT>
std::vector<ResultT> map(const MapperInputT& input);

template<class ReducerInputT, class ResultT>
ResultT reduce(const ReducerInputT &input);
```

La classe `MapperInputT` rappresenta il dato letto dal file (per file testuali, ad esempio, una stringa che rappresenta una linea del file). La classe `ResultT` contiene due attributi: una chiave e un valore (key, result). È utilizzata per salvare i risultati delle funzioni map e reduce. La classe `ReducerInputT` contiene tre attributi: chiave, valore da ridurre, accumulatore corrente (key, value, acc). Rappresenta il dato in input alla funzione reduce.

## Parte 1

Come prima parte dello svolgimento dell'esercizio implementare il paradigma *MapReduce* in modo che lavori con un singolo processo. Il possibile pseudo codice è il seguente:

```
function mapreduce(input, mapfun, reducefun){
  accs = {}
  while line = readline(input){
    results = mapfun(MapperInputT(line))
    for key, result in results {
      new_key, new_acc = reducefun(ReducerInputT(key, result, accs[key]));
      accs[new_key] = new_acc
    }
  }
  return accs;
}
```

Si noti come le funzioni `mapfun` e `reducefun` devono essere passate come parametri alla funzione `mapreduce`. Si implementino quindi le funzioni di map e reduce per:

- contare quante richieste ci sono state per ogni indirizzo IP
- contare quante richieste ci sono state per ogni ora del giorno (ignorando la data)
- contare le url più visitate
- per ogni codice di risultato (200, 201, ..., 404, ..., 500,...) raggruppare gli IP che hanno fatto richieste ottenendo quel codice e stampare quelli che hanno tentato attacchi richiedendo url con errori (codice 400, 404, 405). (Suggerimento: la chiave può essere costruita ad hoc, tipo "codice-ip").

## Parte 2

Il passo successivo consiste nel suddividere il lavoro su tre processi, un coordinare, un mapper e un reducer utilizzando la funzione `fork()` per creare i sotto-processi e collegandoli tra loro tramite delle pipe.
**Riflessione**: Attraverso una pipe è possibile passare direttamente gli oggetti MapperInputT, ReducerInputT o ResultT? Per prima cosa, ci occupiamo di rendere possibile la comunicazione. Con le pipe abbiamo due *file descriptor* e non esiste un out-of-the-box per ottenere da questi degli input e degli output stream C++. Pertanto per comunicare occorre usare le le funzioni di basso livello `write` e `read` e trasformare gli oggetti in sequenze di byte facilmente analizzabili e che permettano di sapere esattamente quando inizia e finisce ogni oggetto e ogni suo attributo.
Le possibilità per fare questo sono molteplici:

- Una codifica usando qualche formato standard, come ad esempio Protocol Buffer[1] o JSON.
- Per JSON è possibile definire un oggetto corrispondente ad ogni parametro e sfruttare il fatto che si possono serializzare in una singola riga, quindi ogni oggetto scritto sulla pipe sarà separato da un newline (il JSON garantisce che dentro il messaggio non vi siano newline).

In questo caso utilizzeremo JSON. Si serializzino quindi i risultati in JSON usando la libreria JSON di boost[2] ottenendo, ad esempio, un array di risultati simile a:

```
[ {"key": "val1", "result": {...}}, {"key": "val2", "result": {...}}, ... ]
```

[1] `https://developers.google.com/protocol-buffers`
[2] `https://github.com/boostorg/json`

Quindi, si possono serializzare gli oggetti che abbiamo utilizzato per salvare i dati e i risultati e i loro contenuti tramite sequenze di byte. Per realizzare la serializzazione, quindi, definiamo per `MapperInputT`, `ReducerInputT` e `ResultT` delle funzioni di serializzazione :

```
std::vector<char> serialize() const;
void deserialize(std::shared_ptr<char*>);
```

Quindi `serialize()` scriverà nel vettore in uscita il contenuto dell'oggetto corrente secondo il seguente formato (*occhio che le lunghezze vanno scritte in binario*):

`<lunghezza totale><array attributi in ordine: <lunghezza attributo> <attributo>>`

**Opzionale**: Si provi ad implementare le funzioni serialize/deserialize come interfaccia usando il Curiously Recurring Template Pattern.

Il passo successivo è quindi quello di introdurre l'utilizzo delle pipe sfruttando la serializzazione definita in precedenza. Per comunicare è possibile usare solo i metodi di basso livello `read` e `write`, che normalmente sono bloccanti e quindi questo meccanismo può essere sfruttato per la sincronizzazione dei processi.. Si implementi quindi il seguente pseudocodice:

```
function mapreduce(input_file, mapfun, reducefun){
  // create mapper_pipe and reducer_pipe
  // fork mapper and reducer
  accs = {}
  while line = readline(input_file){
      // MapperInputT serve per rendere serializzabile line,
      // che presumibilmente sara' una stringa
      write_to_pipe(mapper_pipe, MapperInputT(line))
      results = read_from_pipe(mapper_pipe)
      for key, result in results {
         write_to_pipe(reducer_pipe, MapperInputT(key, result, accs[key]))
         new_key new_acc = read_from_pipe(reducer_pipe)
         accs[new_key] = new_acc
      }
  }
  return accs;
}
```

Un esempio di definizione delle funzioni `read` e `write` sulle pipe:

```
template<class T>
void write_to_pipe(int fd, const T& obj); template<class T>
T read_from_pipe(int fd);
```

Suggerimenti:

- nella `write` usare `reinterpret_cast` per ottenere un `char*` da un vettore
- nella `read` leggete prima la lunghezza dell'oggetto e poi in una seconda read esattamente i byte indicati

**Riflessione**: quale problema presenta questo approccio sincrono in termini di prestazioni?
Infine gestire la terminazione corretta del pattern MapReduce, ricordare che, nell'ordine:

- i figli devono sapere che non ci sono più dati da processare;
- il padre deve sapere quando i figli non hanno più task residui da inviare.

# Parte3

Per sfruttare in modo opportuno il possibile parallelismo, il coordinatore dovrà occuparsi di rifornire il più velocemente possibile di lavoro i due task subordinati, inviando al mapper i dati presenti nel file di ingresso e al reducer i dati via via prodotti dal mapper senza fermarsi ad aspettare i singoli risultati. Poiché i tempi di elaborazione dei singoli compiti non sono prevedibili e poiché le pipe hanno una capacità limitata di contenere dati "in transito", il coordinatore dovrà sforzarsi di leggere, il più frequentemente possibile, da ciascuno dei canali in ingresso dei dati e riversarli nel corrispondente canale di uscita, senza attendere se non vi sono dati o non c'è un messaggio completo.

Occorre però evitare di rimanere bloccati quando il canale di uscita è già "saturo". Di conseguenza, tutte le operazioni di I/O, read e write, devono essere effettuate in modo non bloccante. Si legga attentamente il documento `https://www.geeksforgeeks.org/non-blocking-io-with-pipes-in-c/` allo scopo di capire meglio la natura del problema.
Inoltre vedere in dettaglio il comportamento di write e read non bloccanti sulle pipe: `https://linux.die.net/man/3/write` e `https://linux.die.net/man/3/read`. Come si comportano read e write quando il buffer è vuoto o pieno? Che succede ad esempio se in una read nel buffer vi sono meno byte di quelli richiesti o se è vuoto? Quali modifiche al codice precedente sarebbero necessarie necessarie? (Non implementare codice a questo passo, limitarsi a descrivere i problemi da superare).

Per gestire la complessità dell'IO asincrono si può utilizzare la **select** (o sue implementazioni alternative come pselect, poll o epoll), che permette di testare, prima su quali file descriptor si può leggere e scrivere senza rimanere bloccati, e poi, fare le operazioni di IO solo nelle pipe corrispondenti. In questo modo si possono preservare i benefici di read e write bloccanti, senza pagarne troppo il prezzo in termini di prestazioni. Per il suo utilizzo fare riferimento a questo link: `https://www.tutorialspoint.com/unix_system_calls/_newselect.htm`. Il meccanismo di funzionamento base della select è la ricezione in ingresso un set di file descriptor da osservare (fd_set) e un timeout. La funzione ritorna:

- -1 in caso di errore
- 0 se è scaduto il timeout senza che nessun file descriptor potesse essere usato in modo non bloccante
- un valore maggiore di zero altrimenti. In questo caso fd_set contiene la lista dei file descriptor la cui read e write non è bloccante, e si possono testare con la macro FD_ISSET.

Occorre pertanto riorganizzare il codice del coordinatore scritto in precedenza per fare sì che lettura e scrittura ad ogni ciclo siano opzionali e fatte soltanto se in quel momento la pipe non è bloccante.
Trasformare il codice sincrono precedente scritto, seguendo questo pseudocodice:

```
input_queue = [ ]
result_queue = [ ]
while (true) {
  fd_set = (input, mapper_pipe_in, mapper_pipe_out, reducer_pipe_in,
     reducer_pipe_out)
  ret = select(&fd_set, time_out)
  switch(ret){
  case -1:
    // handle error
    break;
  case 0:
    // nothing to do
    break
```

```
  default:
    if(is_set(input, fd_set) && input_queue.lenhgth < MAX_LEN) {
      // read from input and queue in input_queue
    }
    // test mapper input and write input if any
    // test mapper output, read and queue results
    // test reducer input and write queue results if any // test reducer
       output and read results
}
```