Concorrenza e corse critiche nelle web-applications

Cristiano Longo

4 Aprile 2003

Indice

1 Introduzione **3**

2 Corse critiche e semafori **3**
2.1 Multitasking . . . 3
2.2 Corse critiche . . . 5
2.3 I semafori . . . 6

3 Applicazioni WEB **7**
3.1 I server web ed il protocollo http . . . 8
3.2 Programmazione di applicazioni web . . . 8
3.2.1 CGI . . . 9
3.2.2 Linguaggi di scripting . . . 9
3.3 Server web e problemi di concorrenza . . . 9

4 Caso di studio : implementazione di un Contatore di accessi in PHP **12**
4.1 Progettazione . . . 12
4.2 Files . . . 13
4.2.1 Apertura del file . . . 13
4.2.2 Lettura da file . . . 13
4.2.3 Posizionamento sul file . . . 14
4.2.4 Scrittura file . . . 14
4.2.5 Troncamento del file . . . 15
4.2.6 Chiusura del file . . . 15
4.3 Semafori . . . 15
4.3.1 Ottenere l'identificativo di un semaforo : ftok ed fget . . 16
4.3.2 Operare sui semafori . . . 16
4.4 Implementazione . . . 16
4.4.1 Brevissima introduzione alla programmazione ad oggetti 17
4.4.2 Attributi della classe . . . 17
4.4.3 Costruttore . . . 17
4.4.4 Leggere il valore del contatore . . . 18
4.4.5 Modificare il contatore . . . 19
4.4.6 Rendere definitive le modifiche . . . 19
4.5 Inserire il contatore in una pagina web . . . 20

1 Introduzione

L'avvento della *computazione parallela*(multitasking) ha portato notevoli benefici sia riguardo i tempi di esecuzione, sia riguardo gli schemi e gli stilemi di programmazione. Essa però ha messo la comunità degli informatici di fronte a nuove questioni che fino a quel momento non si erano mai poste(deadlock e corse critiche). La programmazione di web-applications, intese come applicazioni che usino un web-server per interfacciarsi con gli utenti, per la natura stessa dei web-server e del protocollo http devono necessariamente fare i conti con i problemi derivati dall'utilizzo del multitasking. Un web server, difatti, per servire più richieste contemporaneamente, genera un task per ogni richiesta. Nel momento in cui la web-application faccia riferimento ad una qualche entità *unica*(ad esempio un data-base o un file) sorgono i problemi tipici della programmazione parallela.

In questo articolo verranno discussi i problemi relativi alle corse critiche nelle web-applications, e una possibile soluzione basata sui *semafori*. A questo scopo verrà esaminato come caso di studio la realizzazione di un contatore d'accessi per un sito web realizzato mediante il linguaggio di scripting server-side PHP.

La prima parte dell'articolo conterrà una panoramica generale sul problema delle corse critiche e sull'utilizzo dei semafori per prevenirle. La seconda tratterà invece il funzionamento di un web-server e le differenze che esistono tra la progettazione di una applicazione *classica* e di una web-application.

Infine verrà esaminato il problema della realizzazione di un contatore di accessi per siti web, delineando i problemi che nascono a causa della *concorrenza* e una soluzione basata sui semafori.

2 Corse critiche e semafori

2.1 Multitasking

La computazione multitask nasce in contrapposizione alla cosiddetta computazione *batch*, o single-task. Nello schema di computazione batch vari *job* (programmi)venivano inseriti in una coda in un ben determinato ordine ed eseguiti uno per volta. Solo quando un job aveva completato la sua esecuzione poteva iniziare il successivo.

Lo schema multitask prevede invece la possibilità di eseguire molteplici *task*(l'equivalente dei job) contemporaneamente. Dire contemporaneamente in realtà non è del tutto esatto. Nei calcolatori con un unico processore le operazioni in linguaggio macchina vengono ovviamente eseguite una per

volta, motivo per il quale in un dato istante un'unico task(processo) può essere in esecuzione. La contemporaneità viene più che altro simulata. Il tempo di CPU viene suddiviso in *quanti*, da assegnare volta per volta ad ogni singolo processo in esecuzione. All'inizio di un quanto di tempo una entità del sistema operativo, lo *schedulatore*, decide a quale processo assegnare il quanto che sta per iniziare. Il processo designato progredisce nella sua esecuzione fino al termine del tempo assegnato. A questo punto viene sospeso ed il ciclo ricomincia.

Ad una prima occhiata può sembrare che il tempo impiegato per completare n processi eseguiti in parallelo non sia inferiore al tempo necessario per portare a termine l'esecuzione gli stessu n processi eseguiti uno alla volta(come nello schema batch). Anzi lo schema batch risulta più veloce inquanto sul tempo totale non grava il sovraccarico dovuto alla schedulazione e al cambio continuo da un task ad un'altro tipico dei sistemi multitask. Questo è vero finquando i processi da eseguire non hanno la necessità di accedere a risorse *lente*. Supponiamo a titolo di esempio di avere due processi : A e B. Supponiamo che il processo A esegua un alto numero di accessi ad un disco. Le operazioni di I/O su periferiche di solito non richiedono bassi tempi di utilizzo della CPU e lunghi tempi di latenza durante i quali la periferica esegue l'azione richiesta. Questi tempi di latenza, durante i quali in un sistema batch il processore resterebbe inutilizzato, possono essere invece utilizzati dal processo B, riducendo così il tempo totale di esecuzione di A e B.

2.2 Corse critiche

Supponiamo che contemporaneamente siano in esecuzione su un calcolatore n processi $p_1, \dots p_n$. La durata di un quanto di tempo(numero di cicli di clock) e l'algoritmo di schedulazione sono fattori dai quali lo sviluppatore di applicazioni deve poter prescindere. Inoltre l'ordine esatto nel quale le istruzioni di tali processi verranno eseguite non può essere noto a priori.

Supponiamo ad esempio di avere un programma P, che funziona secondo il seguente algoritmo :

1. leggi i dalla risorsa R
2. incrementa i
3. setta il valore di $R = i$
4. ritorna i

Supponiamo che a un dato istante t, R abbia assunto il valore x, e che siano in esecuzione due *copie*(processi) del programma P : p_1, p_2. Se siamo particolarmente sfortunati, l'ordine nel quale le istruzioni di p_1 e p_2 potrebbe essere il seguente :

istante	R	p_1	p_2
t	x	leggi i_1 dalla risorsa R : $i_1 = x$	sleep
$t+1$	x	incrementa i_1 : $i_1 ++ \rightarrow i_1 = x+1$	sleep
$t+2$	x	sleep	leggi i_2 dalla risorsa R : $i_2 = x$
$t+3$	x	sleep	incrementa i_2 : $i_2 ++ \rightarrow i_2 = x+1$
$t+4$	$x+1$	setta il valore di $R = i_1 = x+1$	sleep
$t+5$	$x+1$	setta il valore di $R = i_2 = x+2$	sleep

Al termine dell'esecuzione dei due processi il valore di R, che avrebbe dovuto essere incrementato di due unità, risulta aumentato solo di una. Abbiamo appena visto un esempio di *corsa critica* che ha causato un errore. Il verificarsi di tale errore è stato causato dall'accesso alla risorsa R di un altro processo mentre il primo non aveva ancora ultimato l'operazione. Le operazioni previste dal programma P è necessario che vengano eseguite in maniera *atomica*, ossia senza che chiunque altro abbia la possibilità di accedere ad R nel mentre. I passi 1, 2, 3 del programma P si dice allora che rappresentano una *sezione critica* per l'accesso alla risorsa R, e si incorre in una corsa critica se due sezioni critiche sulla stessa risorsa vengono eseguite in momenti non disgiunti.

2.3 I semafori

Per evitare le intersezione delle sezioni critiche sono state presentate svariate soluzioni. In questo documento prendiamo in esame una soluzione basata sui *semafori binari*. Un semaforo è una struttura messa a disposizione dal sistema operativo, e non potrebbe essere altrimenti considerando l'effetto che ha sull'esecuzione dei processi. Un semaforo binario può trovarsi in due stati : esso può essere *disponibile*, o può essere stato *acquisito* da un processo.

Ogni processo può tentare di acquisire un determinato semaforo. Se tale semaforo però non è disponibile, il processo si blocca finchè il semaforo non viene rilasciato. Una volta che un processo p ha acquisito un semaforo, solo p può rilasciarlo, rendendolo disponibile a tutti gli altri processi.

Supponiamo di associare un semaforo s ad una risorsa R. Se ogni sezione critica che acceda ad R inizia col tentativo di acquisire s e termina col rilascio di questo stesso, possiamo essere certi di non avere intersezioni. L'algoritmo visto in precedenza diventa.

1. acquisisci s
2. leggi i dalla risorsa R
3. incrementa i
4. setta il valore di $R = i$
5. rilascia s
6. ritorna i

Sia ad esempio p_1 all'interno della sezione critica. Abbia egli acquisito s. Supponiamo che p_2 tenti di entrare nella sezione critica. Egli deve innanzitutto acquisire s. Ma s non è disponibile, quindi p_2 si addormenta finchè p_1 non esce dalla sezione critica rilasciando s.

3 Applicazioni WEB

In questi ultimi hanni si è verificato un graduale ritorno alle architetture client-server, molto in voga agli albori della rete internet e progressivamente abbandonato con l'avvento degli home-computers e la grande diffusione di questi. Una architettura client-server offre di certo innumerevoli vantaggi sia sotto il profilo della manutenzione del software(il server può essere aggiornato in continuazione in maniera spesso trasparente ai client) sia a livello di costi. L'innovazione alla quale si assiste oggi però riguarda la struttura dei client. Si inizia ad affermare un nuovo modello basato sui protocolli HTTP e HTML/XHTML che prevede la fruizione del sistema, interamente implementato sul server, tramite un semplice web-browser. I vantaggi sono innanzitutto un abbattimento dei costi e dei tempi dovuti allo sviluppo della GUI grazie all'uso dell'HTML, la reale possibilità di utilizzo del sistema da parte di client di qualsiasi piattaforma (che fornisca la possibilità di navigare su internet tramite un browser) ed infine il fatto che il mantenimento e lo sviluppo del software possono essere totalmente realizzati lato server senza doversi preoccupare dei browser.

Tuttavia, come vedremo in dettaglio più avanti, le applicazioni che comunicano con i clienti tramite i browser devono farlo mediante il protocollo HTTP, e quindi è necessario che si appoggino in qualche modo ad un web-server(già esistente o realizzato ad hoc). Questo porta a dover strutturare la programmazione delle applicazioni di questo tipo in maniera radicalmente diversa da come viene strutturata una normale applicazione client-side, introducendo nuove problematiche e portando all'estremo limite alcune di quelle preesistenti, come avviene per l'accesso concorrente a risorse condivise.

3.1 I server web ed il protocollo http

Lo scopo di un server web è rendere fruibili attraverso internet blocchi di informazioni, detti *pagine*, residenti sul server stesso ad utenti remoti. Il protocollo HTTP, che regola l'interazione tra il client ed il server web, funziona secondo il seguente schema.

1. il client apre una connessione TCP col server.
2. il client richiede una pagina, attraverso un comando del tipo

   GET nome della pagina

3. il server invia la pagina
4. il server chiude la connessione

Questo basta a farci comprendere quanto sia diversa la logica necessaria per realizzare una applicazione client-side, che in ogni momento conosce il proprio stato interno che e' il risultato di una sequenza di azioni compiute dall'utente, rispetto a quella di una applicazione web, le cui uniche informazioni sullo stato di base sono ricavabili dall'azione di richiesta di una pagina.

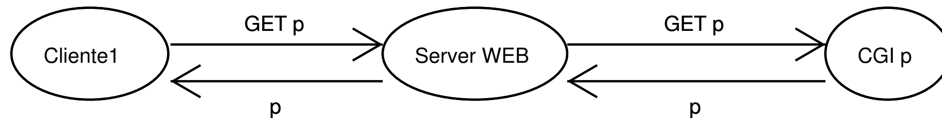
3.2 Programmazione di applicazioni web

La definizione di *applicazione web*, nel senso proposto in questo documento, prevede che l'interfaccia utente(l'output dell'applicazione) sia realizzata mediante HTML. Le *pagine* indicate nella sezione precedente non sono altro che, di norma, documenti testuali in formato HTML. Lo scopo di un browser è di prelevare questi documenti dal server e *renderizzarli*(mostrarli con una certa formattazione ed un certo aspetto) sullo schermo. Quindi una applicazione web produce in output fondamentalmente documenti HTML.

Le modalità più diffuse per la realizzazione di questo tipo di applicazione sono la creazione di *moduli binari CGI* e l'utilizzo di *linguaggi di scripting*.

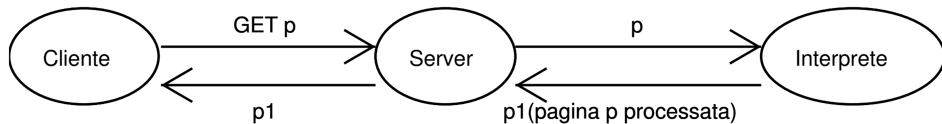
3.2.1 CGI

Un CGI è un file binario eseguibile. Ad ognuno di questi è demandata la creazione di una pagina. Quando tale pagina viene richiesta il web server invoca il CGI corrispondente passando la richiesta come parametro. Questi ritorna un documento HTML che il server utilizza per soddisfare la richiesta.



3.2.2 Linguaggi di scripting

Un altro approccio consiste nell'utilizzo di un *interprete* unico per processare non le richieste bensì le pagine. Questo approccio permette di *immergere* del codice scritto nel linguaggio dell'interprete all'interno del codice html. In questo schema il server web, se si rende conto che la pagina richiesta deve essere processata, la passa all'interprete. Questo provvede a processarla e a restituire una documento html.

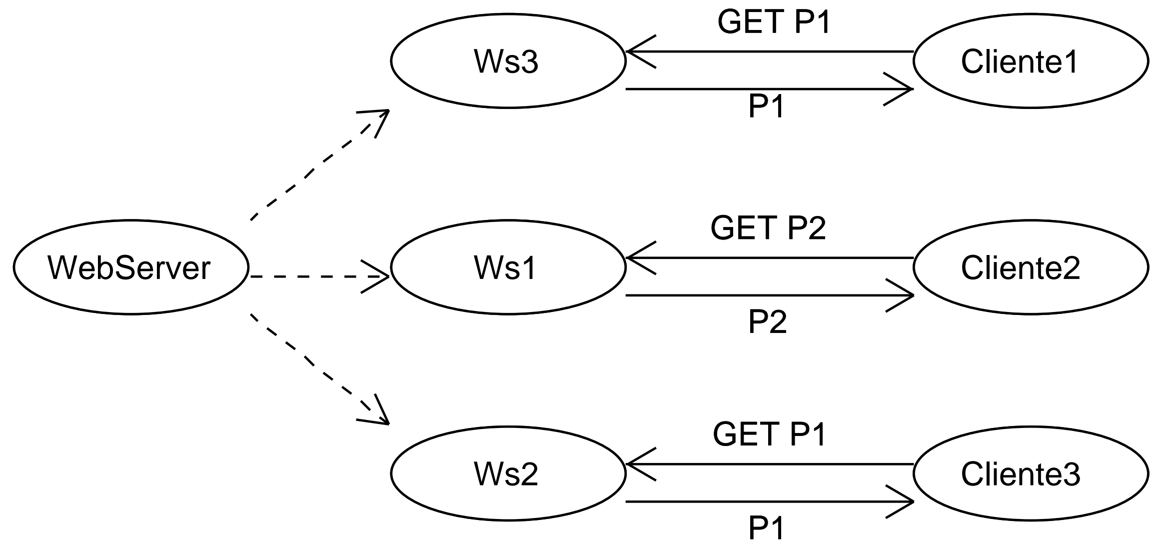


3.3 Server web e problemi di concorrenza

Ma vediamo in dettaglio il comportamento di un web server che si trovi a dover servire più richieste contemporaneamente, per comprendere perchè nascono i problemi relativi alla concorrenza per l'accesso a risorse condivise. Per poter servire più di una richiesta il server web, quando riceve una richiesta di connessione sulla porta 80(di default) si *replica*, creando un processo identico al padre che però avrà una vita propria da quel momento in poi. A questo punto una nuova porta tra quelle disponibili superiori alla 1024 viene assegnata a questo nuovo processo ed il cliente viene servito su questa porta da questo nuovo processo.

Per questo motivo l'accesso a qualsiasi risorsa presente sul server può avvenire in contemporanea da diverse *repliche* del web server che servano

richieste per la stessa pagina o per pagine differenti che richiedono l'accesso alla stessa risorsa. Si rende quindi necessario analizzare e gestire l'eventuale verificarsi di corse critiche ogni qual volta una pagina acceda ad una risorsa condivisa sul server.


GET P1
Ws3
Cliente1
P1
GET P2
WebServer
Ws1
Cliente2
P2
GET P1
Ws2
Cliente3
P1

4 Caso di studio : implementazione di un Contatore di accessi in PHP

Esaminiamo ora a titolo di esempio la realizzazione di un *contatore di accessi per pagine web*. Come già messo sottolineato nella sezione 2.2 una applicazione di questo tipo può portare al verificarsi di corse critiche se eseguita da due o più processi in contemporanea. Vedremo che le soluzioni proposte continuano ad essere valide anche nel caso di WebApplication.

4.1 Progettazione

Un contatore di accessi per una pagina web non è altro che un *numero* che viene incrementato ogni volta che la pagina viene visitata. Esso può essere quindi schematizzato come un *oggetto* che contiene al suo interno un valore numerico, e che rende disponibili all'esterno due *metodi* : *zero* per azzerare il valore del contatore, *read* per leggere il valore corrente e *inc* per incrementare il contatore.

```
class Counter{
  void zero(){ ... }
  int read() { ... }
  void inc() { ... }
}
```

Per l'*implementazione*, per evitare l'utilizzo di data-bases e altre strutture complesse, conserveremo il valore del nostro contatore in un *file* residente sul server. Mentre per la lettura del contatore non sussistono problemi di concorrenza, per quanto riguarda l'incremento sarà necessario regolare l'accesso al contatore per evitare i problemi sottolineati nel paragrafo 2.2. Per fare questo, verrà implementata la soluzione proposta al paragrafo 2.3, basata sui semafori.

Verrà quindi innanzitutto mostrato, a scopo preparatorio, come gestire files e semafori nel linguaggio PHP.

4.2 Files

Le funzionalità offerte da PHP riguardo la gestione dei files richiamano anche nel nome le funzioni dedite a questo scopo nel linguaggio ANSI C(in particolare nella libreria stdio). Verranno esaminate in questo paragrafo le funzioni che poi verranno utilizzate per l'implementazione del contatore.

4.2.1 Apertura del file

Innanzitutto è necessario *aprire* il file. Questo è possibile grazie alla funzione

```
int fopen ( string filename, string mode [, int use_include_path])
```

Il primo argomento, `filename`, può essere del tipo `http://SERVER` o `ftp://SERVER`. In questo caso verrà aperta una connessione del rispettivo protocollo col `SERVER` indicato. Nel caso in cui `filename` non abbia nessuna delle due forme, viene aperto il file da esso indicato sul filesystem locale. Il secondo argomento indica la *modalità* di apertura del file. Tra le modalità consentite troviamo w(scrittura), r (lettura), ... La trattazione del terzo argomento non rientra negli scopi di questo documento. Il valore ritornato è un *descrittore* di file, o la costante `FALSE` nel caso si sia verificato un errore.

4.2.2 Lettura da file

Il file utilizzato per contenere il valore del contatore avr come contenuto unicamente un *numero intero* rappresentato come *stringa*. La rappresentazione del numero in formato di stringa ci permette di evitare le problematiche relative alla dimensione dell'intero che nascerebbero con una rappresentazione binaria. Essa ci permette inoltre di introdurre una interessante funzione di manipolazione delle stringhe :

```
mixed fscanf ( int handle, string format [, string var1])
```

che rappresenta per i file l'analogo di `scanf`. Il primo argomento è un descrittore di file ottenuto mediante `fopen`. Il secondo è invece una *stringa di formattazione*. Essa può contenere dei *segnaposto* intervallati da sequenze di stringhe. I segnaposto indicano dove leggere e come interpretare i valori da assegare alle variabili che saranno date in output. I segnaposto più comuni sono `%s` per indicare stringhe e `%d`(che utilizzeremo nel seguito) per indicare interi. Ad esempio la stringa

```
Mario ha 19 anni
```

letta con scanf da un file mediante il formato

```
%s ha %d anni
```

restituirà due valori : una *stringa* contenente la parola `Mario` e l'intero `18`. La funzione `fscanf` ritorna un vettore contenente i valori indicati dalla stringa di formato.

> Attenzione, l'utilizzo di fscanf al posto di fread, sicuramente più indicato, ha uno scopo puramente didattico.

4.2.3 Posizionamento sul file

Dopo aver letto il contatore, l'unica cosa che resta da fare è scriverne il valore incrementato di uno. E' tuttavia necessario prima riportare il puntatore del file all'inizio del file stesso. Dopo ogni operazione, quindi anche una lettura, il puntatore del file viene *spostato* del numero di posizioni interessate. Ad esempio dopo la lettura n byte, il puntatore si trova sul byte n+1. Ogni operazione, a questo punto, viene effettuata da questa posizione in poi. Per spostare il puntatore si usa la funzione

```
int fseek ( int fp, int offset [, int whence])
```

Questa sposta il puntatore del file `fp` alla posizione `whence` + `offset`. I valori che può assumere `whence` sono `SEEK_SET`, che indica l'inizio del file, `SEEK_CUR`, per la posizione attuale, e `SEEK_END` che indica la fine del file. Se `whence` non viene specificato, assume il valore di default `SEEK_SET`. Per riportare il puntatore all'inizio del file, sarà sufficente quindi invocare

```
fseek (fp, 0);
```

Esiste tuttavia una *scorciatoia* per ottenere lo stesso risultato, la funzione `rewind(fp)` che riporta il puntatore all'inizio del file.

4.2.4 Scrittura file

Dopo essersi riposizionati all'inizio del file, è necessario scrivere il nuovo valore. Questo è possibile farlo con la funzione `fprintf`(simile a `fscanf`) o mediante

```
int fwrite ( int fp, string string [, int length])
```

che scrive la stringa `string` sul file `fp`.

4.2.5 Troncamento del file

Nel caso di azzeramento, dopo la scrittura, è necessario troncare il file, per eliminare eventuali cifre *residue*. Supponiamo ad esempio che il contatore abbia assunto il valore 123123. Sul file troveremmo la stringa

```
123123
```

A questo punto viene chiamata la funzione di azzeramento. Il puntatore viene riportato all'inizio e la stringa `0` viene scritta sul file. La situazione sul file sarà la seguente :

```
023123
```

perchè le cifre oltre la prima non sono stato rimosse. Per eseguire questo compito è necessario invocare la funzione

```
int ftruncate ( int fp, int size)
```

che *tronca* la dimensione del file alla dimensione indicata da `size`

4.2.6 Chiusura del file

Non resta quindi che chiudere il file e rendere definitive eventuali modifiche.

```
bool fclose ( int fp)
```

4.3 Semafori

È stato deciso, a scopo puramente didattico, di realizzare questa implementazione del contatore mediante semafori per eliminare il pericolo di corse critiche. In realtà esiste un metodo più diretto per ottenere questo scopo. L'interprete PHP offre di fatti la possibilità di *lockare* un file mediante le primitive *lock* e *unlock*. Tali primitive tuttavia si basano sui *semafori binari* offerti dal sistema operativo. Motivo per il quale studiare l'utilizzo di tali semafori, oltre a poter essere utile anche in circostanze diverse, ci aiuta meglio a capire il meccanismo mediante il quale un file viene *lockato*.

4.3.1 Ottenere l'identificativo di un semaforo : ftok ed fget

Il sistema per la gestione di risorse condivise è stato ereditato(come molto altro) dalle architetture **Sistem V**. Tale sistema è disponibile solo su sistemi operativi UNIX-LIKE, motivo per il quale questa implementazione **non funziona su server Microsoft**. SistemV consente di identificare univocamente ogni risorsa condivisa mediante opportune *chiavi* ottenibili con la primitiva

```
int ftok ( string pathname, string proj )
```

Ove `pathname` rappresenta il path della risorsa condivisa(nel nostro caso il nome del file) e `proj` un identificativo aggiuntivo di una sola lettera che ha lo scopo di permettere ulteriori differenziazioni.

Ottenuta la chiave SistemV, possiamo ottenere l'identificativo del semaforo mediante la primitiva

```
int sem_get ( int key [, int max_acquire [, int perm]])
```

Il secondo parametro indica quanti processi possono acquisire contemporaneamente il semaforo(di default uno), ed il terzo specifica eventuali permessi sul semaforo stesso, secondo la notazione UNIX.

4.3.2 Operare sui semafori

Le primitive che permettono di acquisire e rilasciare un semaforo sono rispettivamente

```
bool sem_acquire ( int sem_identifier)
bool sem_release ( int sem_identifier)
```

Nel caso in cui un processo invochi `em_acquire*␣mentre␣il␣`emaforo è stato già acquisito da un altro processo, ma non ancora rilasciato, questo si *addormenta*, per *svegliarsi* non appena il semaforo diventa disponibile ed acquisirlo a sua volta.

4.4 Implementazione

Abbiamo finalmente gli strumenti necessari per la realizzazione del nostro contatore di accessi in PHP. Questa fase offre l'occasione per introdurre una importante funzionalità offerta dal linguagio PHP : la possibilità di utilizzare il paradigma della *programmazione ad oggetti*.

4.4.1 Brevissima introduzione alla programmazione ad oggetti

Questo paradigma fu accolto con entusiasmo(ed anche un pò di diffidenza) dagli sviluppatori poichè permette di modellare il processo di sviluppo del software sugli oggetti e le entità che sono realmente coinvolti o devono essere simulati nell'utilizzo finale del software stesso.

Il paradigma della programmazione ad oggetti prevede due concetti fondamentali: quello di classe, e quello di istanza(oggetto). Una *classe* rappresenta una *descrizione*, a partire dalla quale è possibile costruire un insieme potenzialmente infinito di *oggetti*, tutti però con le stesse caratteristiche(definite appunto nella classe).

Prendiamo come esempio le automobili. Ogni automobile deve avere, per essere definita tale, quattro ruote, un telaio, il cambio, ... Tutte queste caratteristiche definiscono la classe automobile. Questa classe ha però varie istanze : la mia macchina, la macchina del vicino, ... tutte rispondenti alla descrizione vista prima, ma fisicamente distinte e indipendenti.

Le *caratteristiche* comuni descritte nelle classi si dividono in *attributi*, che nelle istanze assumono dei valori, e *metodi*, che rappresentano le funzionalità degli oggetti di quella classe. I metodi eseguono delle computazioni e modificano il valore degli attributi dell'oggetto.

Prendiamo, per restare in tema con l'esempio precedente, il cambio di una macchina. Esso ha un attributo, la `marciaCorrente`, e un metodo `cambiaMarcia(x)` che modifica la marcia corrente.

Un oggetto viene istanziato a partire da una classe mediante il *costruttore*. Questo è un metodo che ha lo stesso nome della classe.

4.4.2 Attributi della classe

Definiamo quindi quali attributi deve avere il nostro contatore :

value che contiene il valore del contatore.

fp per l'handler del file che contiene il contatore.

sem_id che rappresenta il semaforo dedicato.

4.4.3 Costruttore

Come detto prima, il costruttore è un metodo con lo stesso nome della classe. Nel nostro caso il costruttore si occupa di aprire il file e, se non esiste, crearlo. Questi inoltre acquisisce il semaforo, decretando l'ingresso nella *sezione critica*.

```
function SemCounter($fileName="default.count"){
  /**
   * Apertura del file.
   * NB:se il file non esiste, deve essere creato.
   **/
  if (file_exists($fileName)) $this->fp=fopen($fileName, "r+");
  else $this->fp=fopen(fileName, "w+");

  //Acquisizione del semaforo
  $this->sem_id=sem_get(ftok($fileName, "c"));
  sem_acquire($this->sem_id);
}
```

Il costruttore ammette alpiù un argomento, il nome del file. Questo permette di utilizzare anche più di un contatore(contemporaneamente o separatamente) per scopi diversi.

Una peculiarità del costruttore è la possibilità di fornire valori di default per i parametri. Se il costruttore viene infatti invocato senza parametri, il parametri `fileName` assume il valore indicato `default.count`.

Altra cosa da notare è che l'accesso agli attributi dell'oggetto avviene nella forma $this-¿attributo. La notazione php impone infatti che, per accedere ad attributi o metodi di un oggetto, si debba usare la forma `Oggetto->metodo/attributo`. La parola chiave `this` indica sempre *l'oggetto corrente*.

4.4.4 Leggere il valore del contatore

Per ottenere il valore del contatore da file si invoca il metodo `read()`. Questi effettua il rewind del puntatore del file, e legge il valore dal file. Si presuppone che il file contenga una stringa del tipo

```
Valore del contatore V
```

Ove `V` è il valore che assume correntemente il contatore. Dopo averlo letto, questo valore viene posto nell'attributo `value`. Infine questo valore viene ritornato.

```
function read(){
  rewind($this->fp);
  fscanf($this->fp,"Valore del contatore %d", $r);
  $this->value=intval($r);
  return $r;
}
```

Poichè l'attributo `value` è numerico, mentre la variabile `r` assume il valore di una stringa, è necessario convertire `r` mediante la funzione `intval`.

4.4.5 Modificare il contatore

I metodi previsti per modificare il contatore sono i seguenti

```
function zero(){
  $this->value=0;
  return "0";
}

function inc(){
  $this->read();
  $this->value++;
  return $this->value;
}
```

Il primo azzera il contatore, mentre il secondo lo incrementa di una unità. Poichè per effettuare questa operazione è necessario conoscere il valore del contatore, il metodo `inc()` invoca `read()` a questo scopo.

4.4.6 Rendere definitive le modifiche

Si può notare che sia `inc()` che `zero()` non scrivono sul file, ma si limitano a modificare il valore dell'attributo `value` e ritornarne il nuovo valore. Le modifiche vegono rese effettive dalla seguente funzione :

```
function end(){

  // il valore del contatore viene scritto sul file
  rewind($this->fp);
  $s="Valore del contatore ".$this->value;
  fwrite($this->fp, "Valore del contatore ".$this->value);
  ftruncate($this->fp, strlen($s));

  //il file viene chiuso
  fclose($this->fp);
  $this->fp=FALSE;

  //ed il semaforo rilasciato
  sem_release($this->sem_id);
}
```

4.5 Inserire il contatore in una pagina web

Vediamo ora come si può procedere per inserire il contatore in una o più pagine web. Innanzitutto è necessario creare un file contenente la classe che rappresenta il contatore, allo scopo di poterlo richiamare da diverse pagine.

```
file SemCounter.php

<?
class SemCounter{

 //valore assunto dal contatore
 var $value;

 //handler del file che contiene il contatore
 var $fp;

 //id del semaforo dedicato
 var $sem_id;
```

```
function SemCounter($fileName="default.count"){
 /**
  * Apertura del file.
  * NB:se il file non esiste, deve essere creato.
  **/
 if (file_exists($fileName)) $this->fp=fopen($fileName, "r+");
 else $this->fp=fopen($fileName, "w+");

 //Acquisizione del semaforo
 $this->sem_id=sem_get(ftok($fileName, "c"));
 sem_acquire($this->sem_id);
}

function read(){
 rewind($this->fp);
 fscanf($this->fp,"Valore del contatore %d", $r);
 $this->value=intval($r);
 return $r;
}

function zero(){
 $this->value=0;
 return "0";
}

function inc(){
 $this->read();
 $this->value++;
 return $this->value;
}

function end(){

 //il valore viene scritto sul file
 rewind($this->fp);
 $s="Valore del contatore ".$this->value;
 fwrite($this->fp, "Valore del contatore ".$this->value);
 ftruncate($this->fp, strlen($s));
```

```
  //il file viene chiuso
  fclose($this->fp);
  $this->fp=FALSE;

  //il semaforo viene rilasciato
  sem_release($this->sem_id);
 }
}
?>
```

È necessario richiamare questa classe da ogni pagina nella quale si vuole utilizzare il contatore. Questo è possibile farlo grazie alla primitiva `include`. Questa prende in input il nome di un file(locale o remoto). Tale file viene letteralmente sostituito alla funzione in fase di preprocessing. Motivo per il quale, inserire `include("SemCounter.php");` in una pagina php equivale ad avere l'intero contenuto di SemCounter.php nella pagina stessa. Una volta caricata la definizione della classe, è necessario crearne un'istanza. La primitiva `new Costruttore(argomenti)` crea una nuova istanza della classe e ne invoca il costruttore. Dopo questo possiamo eseguire le operazioni che vogliamo e mostrare il valore del contenuto. Infine, tali modifiche vengono rese effettive invocando il metodo `end()`.

Mostriamo, a titolo di esempio, una pagina che mostra il valore di un contatore che viene incrementato ad ogni visita.

```
file prova.html

<? include("SemCounter.php");
   $contatore=new SemCounter("prova.count");
?>
<html>
 <head>
  <title>prova di SemCounter</title>
 </head>
 <body>
  <P>Il valore del contatore <? contatore.inc(); ?></P>
  <P><A href="prova.html">Aumenta il valore del contatore</A></P>
 </body>
</html>
```