ISSN 2039-7941
Anno 2018_Numero 399

rapporti tecnici INGV

Realizzazione di un sistema operativo Client-Server per la gestione di stazioni geomagnetiche remote e relativo simulatore in ambiente Unix

399

Istituto Nazionale di Geofisica e Vulcanologia

Segreteria di Redazione
Francesca Di Stefano - Referente
Rossella Celi
Tel. +39 06 51860068
redazionecen@ingv.it

in collaborazione con:
Barbara Angioni (RM1)

REGISTRAZIONE AL TRIBUNALE DI ROMA N.173 | 2014, 23 LUGLIO

Rappresentante legale: Carlo DOGLIONI
Sede: Via di Vigna Murata, 605 | Roma

ISSN 2039-7941
Anno 2018_Numero 399

rapporti tecnici

INGV

REALIZZAZIONE DI UN SISTEMA OPERATIVO CLIENT-SERVER PER LA GESTIONE DI STAZIONI GEOMAGNETICHE REMOTE E RELATIVO SIMULATORE IN AMBIENTE UNIX

Antonino Sicali, Alfio Amantia, Pasqualino Cappuccio

INGV (Istituto Nazionale di Geofisica e Vulcanologia, Sezione di Catania - Osservatorio Etneo)

399

Come citare: Sicali A., Amantia A., Cappuccio P., (2018). Realizzazione di un sistema operativo Client-Server per la gestione di stazioni geomagnetiche remote e relativo simulatore in ambiente Unix. Rapp. Tec. INGV, 399: 1-52.

Indice

1. Introduzione

L'unità di acquisizione remota del sistema *Mag-Net* [Del Negro et al, 2002] impiega una motherboard in formato *PC/104* compatibile con lo standard *IBM* (*ISA*), per questo può utilizzare ed eseguire i sistemi operativi di un normale PC. Fin dall'inizio è stato utilizzato l'*MS-DOS* che grazie alla sua semplicità ha permesso di risolvere molti problemi di stabilità però lasciando al software applicativo di gestione l'onere di gestire i diversi task. Ogni task si autoregola permettendo agli altri di funzionare molto similmente a ciò che accade in un sistema operativo di tipo preemptive (senza pre-rilascio) chiamato anche cooperative multitasking. Ciò comporta non pochi problemi soprattutto nella ripartizione delle risorse. Accade molto spesso che un task ostacoli un altro perchè viene sovraccaricato di lavoro. Si è deciso di ricercare una soluzione al problema della ripartizione non equa delle risorse sostituendo l'*MS-DOS* con un sistema operativo costruito ad *hoc* per un sistema di monitoraggio nello specifico per il sistema *Mag-Net*, multiprogrammato e con rilascio di risorse. La progettazione del nuovo sistema deve prescindere dall'*hardware* che può negli anni cambiare e divenire obsoleto, come è già successo più volte per i *PC/104*.

L'utilizzo di un'architettura *IBM* ha permesso nel corso degli anni di velocizzare lo sviluppo software che poteva avvenire su un normale *PC* che diveniva un vero e proprio laboratorio di test per il sistema di gestione *Mag-Net*. La progettazione del nuovo sistema operativo deve poter preservare quanto più possibile tale vantaggio permettendo di velocizzare lo sviluppo utilizzando un normale *personal computer*. Per questo si è deciso di impiegare un ambiente simulato che sostituisse l'*hardware* durante lo sviluppo. Il 90% del codice sarà realizzato (come già successe in *Mag-Net*) in *C/C++* [Sicali et al, 2016] e verrà completamente provato nell'ambiente simulato. Solo il 10% del software sarà dipendente dall'*hardware*, dovrà essere scritto in *ASSEMBLY* e provato direttamente sull'hardware. Utilizzare il *C/C++* ha permesso di riutilizzare la totalità del software già scritto per il sistema *Mag-Net*.

2. Il sistema *Mag-Net*

Attualmente il software del sistema *Mag-N*et è strettamente sequenziale e mantiene quattro *task* generando una schedulazione senza pre-rilascio (*cooperative multitask*), ovvero ogni *task* decide quanto dovrà utilizzare la CPU e le altre risorse. In futuro, impiegando la multiprogrammazione, si vuole introdurre un sistema operativo con pre-rilascio (*preemptive multitasking*), permettendo così una condivisione equa della *CPU* e non a discrezione dei singoli *task*. Ciò che il software di una stazione magnetica, ma in generale di un sistema di acquisizione, deve compiere può essere spiegato riferendosi al classico problema del produttore e del consumatore [Arpaci-Dusseau et al, 2014]: uno strumento di misura, il magnetometro, produce misure mentre una periferica (*GSM*, *IrDA*, *Storage*, etc) provvede al consumo inviandole ad un centro di raccolta dati. Finora il problema è stato affrontato pensando in maniera sequenziale. Utilizzando un sistema operativo multiprogrammato e facendo uso degli oggetto tipici degli ambienti multitask, come l'*IPC* (*Inter process communication*) e lo *scheduler*, si possono superare i problemi tipici di un ambiente sequenziale, come lo spreco di risorse e la mancanza di equo trattamento. Nel nuovo sistema si avranno tanti moduli indipendenti quante sono le funzionalità della stazione magnetica. Ciascun modulo sarà indipendente dagli altri e non influenzerà minimamente il comportamento altrui. La scalabilità del sistema se ne avvantaggerà permettendo una dinamicità e flessibilità non consentita attualmente. Si potranno aggiungere o togliere funzionalità senza creare iterazioni negative nell'esecuzione degli altri moduli. Scegliendo un *file system* opportuno e implementando il miglior tipo di dato astratto (*ADT, Abstract Data Type*) si riduranno gli sprechi energetici e l'impatto di errori sul disco, ottenendo un miglioramento della stabilità e dell'efficienza [Sicali et al, 2016]. Un sistema operativo specifico e non generico, permetterà di ridurre i task ad una manciata di linee di codice riducendo anche il numero di errori introdotti (*bugs*) ed il tempo di sviluppo/aggiornamento del software applicativo.

3. Progettazione del nuovo sistema operativo

L'architettura che verrà impiegata è quella client-server che permette di dividere in modo netto una porzione del software dall'altra riducendo allo stesso tempo le dimensioni e la complessità del nucleo (*kernel*). La suddivisione in moduli permette inoltre di modificare solamente la porzione di codice che riguarda i canali di comunicazioni tra i processi per ottenere un sistema operativo versatile, impiegabile su qualunque macchina e creando di conseguenza una modalità di simulazione e di sviluppo. Come si vede

dalla figura 1 i moduli sono del tutto indipendenti. Ogni modulo fornisce un proprio contributo all'interfaccia a messaggi che utilizzano tutti i processi.

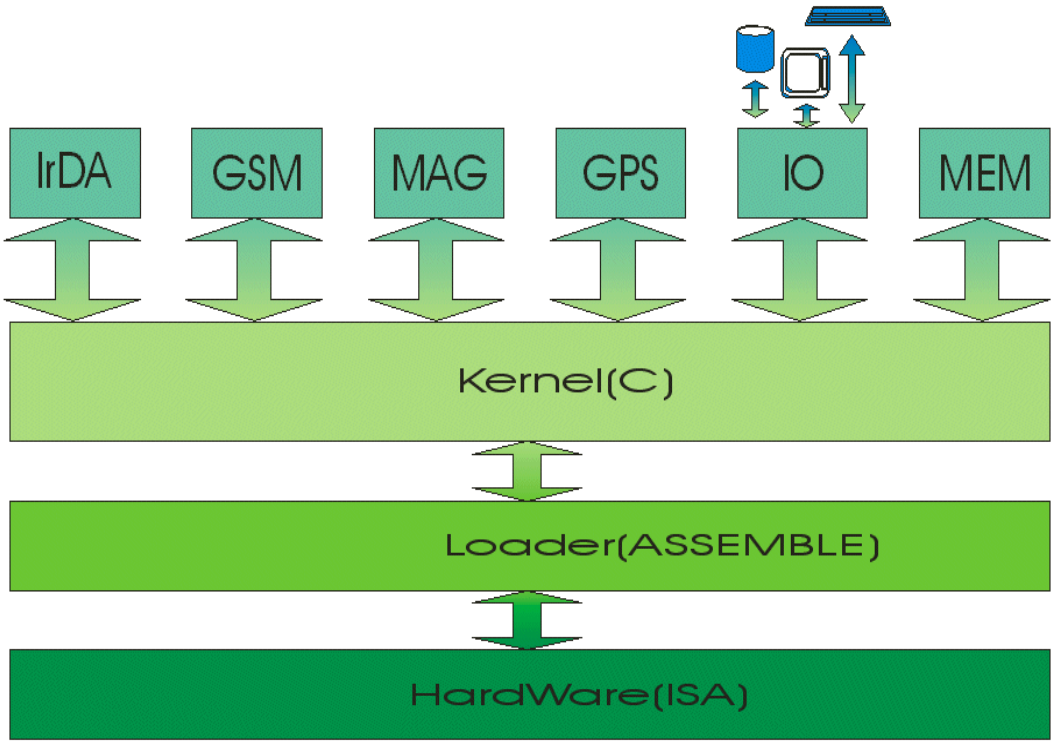

Figura 1. Architettura di *OSGM*.

3.1 Architettura di OSGM (Operating System GeoMagnetism)

L'interfaccia una volta stabilita non è condizionata dai cambiamenti interni a ciascun modulo. Più o meno quello che succede nella programmazione ad oggetti, in cui un oggetto (modulo) può essere modificato e ottimizzato, senza che il resto del programma ne risenta, finchè l'interfaccia con l'esterno rimane inalterata. Il sistema completo comprende diversi moduli, alcuni clienti (*client*) ed altri servitori (*server*) come mostrato in tabella 1.

Nome modulo	Tipo
GPS	Server
Magnetometro	Client (produttore)
IrDA	Client (consumatore)
GSM	Client (consumatore)
IO	Server
Memory	Server

Tabella 1. Processi necessari e tipologia associata.

Ovviamente oltre i moduli descritti esiste anche il nucleo (*kernel*) che si occupa di gestire l'intero sistema ed esplica i compiti previsti per lo *scheduler*, il *messager* e il *file system*. L'esigua quantità di *RAM* utilizzata dai processi, non rende conveniente lo sviluppo di una unità di gestione della memoria fisica (*MMU*) e di modelli per la gestione della paginazione, segmentazione o swapping.

Il software comprende una parte dell'interfaccia a messagi con cui i moduli comunicano, una parte che permette la simulazione su una piattaforma *Unix* (terminale e *X Window*) ed di una parte per creare un funzionamento simulato realistico. Non comprende invece quelle parti che dipendono strettamente dall'*hardware* e quelle parti che saranno riutilizzate dalll'attuale software di gestione del sistema *Mag-Net*, ovvero i moduli finali di comunicazione *(GSM)*, di sincronizzazione temporale (*GPS),* e di acquisizione dal magnetometro.

3.2 Funzionamento

Il simulatore può avviarsi in modalità grafica (figura 2) visualizzando le informazioni in una finestra di *X* Window [Scheifler et al, 1996], oppure in modalità nascosta molto utile durante la fase finale del *debugging* e *test* su lunghi tempi e in configurazione da campagna. Per incrementare il numero di sistemi compatibili la parte grafica può utilizzare sia la libreria *Motif* [Fountain et al, 2001] che *Athena* [MIT, 1983], e permette di monitorare l'esecuzione dei processi, l'occupazione della memoria *RAM* e l'utilizzo del *file system*.

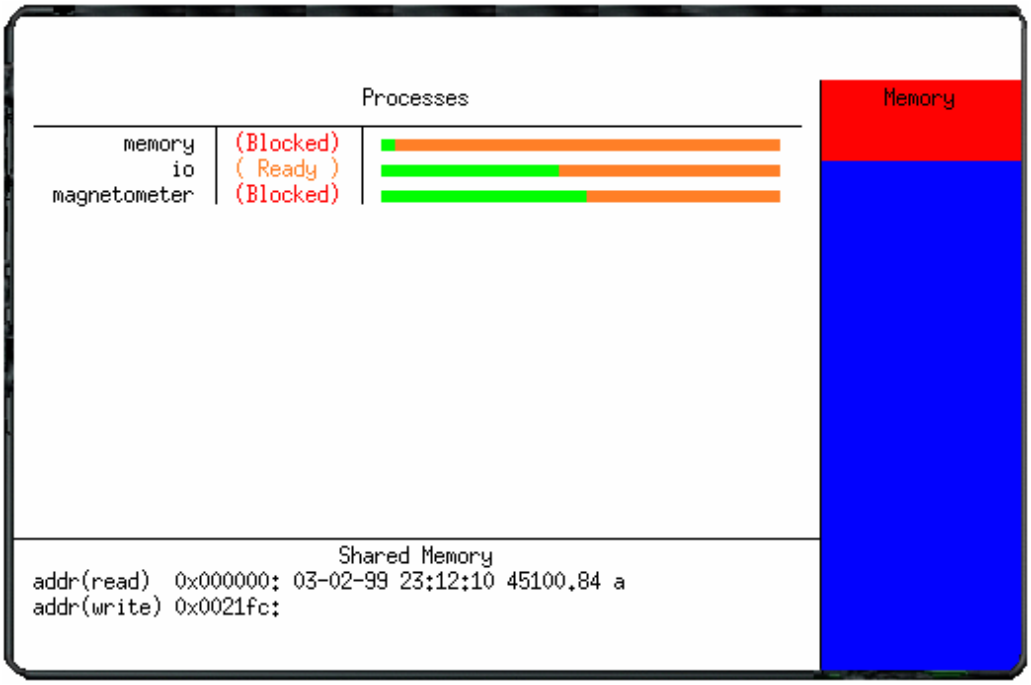

Figura 2. Simulatore in modalità grafica utilizzante le librerie *Athena*.

La comunicazione principale tra i processi viene realizzata impiegando le *pipe* di *Unix*: vengono spediti messaggi di lunghezza arbitraria. Di supporto vengono utilizzati anche altre primitive di comunicazione e sincronizzazione tra processi e *threads* come semafori, memoria condivisa e segnali.

Per avviare il simulatore in modalità grafica si dovrà eseguire il programma contenuto nel file *xosgm* mentre per aviarlo in modalità nascosta si potrà eseguire semplicemente il programma contenuto nel file *kernel*. Il programma contenuto in xosgm dipende dalla compilazione effettuata: per default la compilazione viene eseguita per la libreria grafica *Athena*. Per utilizzare la libreria *Motif* bisognerà eseguire in successione:

```
make; make motif
```

3.2.1 Simulatore: Xosgm

Il programma principale (il simulatore) crea una finestra grafica (figura 2) utilizzando le librerie grafiche di *X* e supportandosi con *XToolkit*, *Motif* oppure *Athena*. Nella finestra grafica verranno visualizzati, ogni qualvolta verranno modificati, i parametri di funzionamento del simulatore. Dopo la creazione della finestra il processo avvia un secondo thread che si occuperà esclusivamente della comunicazione con il sistema operativo in esecuzione, ricevendo attraverso una pipe le notifiche inviate dai processi che compongono il simulatore. Il thread principale invece continuerà la gestione dell'interfaccia grafica. Si è scelta la soluzione del doppio thread per scaricare sul sistema operativo la gestione dei due task che altrimenti dovevano essere gestiti in modo sequenziali all'interno di un ciclo *while* infinito. Per aggiornare le informazioni della finestra grafica il processo utilizza prevalentemente le proprie strutture dati. L'unica struttura dati che utilizza e di cui non ha la proprietà, contiene le informazioni riguardanti la memoria *RAM*. Per accedera alla lista dei blocchi della *RAM*, struttura dati condivisa, si utilizza un semplice semaforo.

3.2.2 Kernel

Il programma contiene le funzionalità offerte dal *kernel* di *OSGM* e verrà eseguito come processo padre di tutti i moduli che gireranno nel nuovo sistema operativo. La lista dei moduli sono forniti al programma elencandoli in un file nella directory corrente chiamato *modules*. Ogni nome di modulo occuperà nel file di testo un'unica riga. Attualmente i moduli caricati sono soltanto *memory*, *magnetometer* e *io*. Una volta avviati i processi figli, questi comunicheranno attraverso delle pipe con il kernel: i moduli useranno un'unica pipe per spedire messaggi al nucleo mentre il nucleo utilizzerà un'intero array di pipe per spedire messaggi ai singoli moduli. Ogni qualvolta un messaggio viene inviato al kernel, un processo genera un segnale *SIGMESS* che attiverà il messager. Si è utilizzata una variabile condivisa tra tutti i processi per evitare che la pertida o l'accorparpamento di segnali possano ingannare il messager. Ogni volta che un processo spedisce un messaggio, la variabile viene incrementata di un'unità, effettuando un operazione *up* sul semaforo con cui è stata implementata la variabile. La procedura che un processo deve rispettare quando manda un messaggio è molto complessa e non verrà mai gestita direttamente dai moduli che invece potranno chiamare la system call *sendMessage*. Di system call ne esistono diverse ed in futuro se ne potranno aggiungere altre. Le system call attualmente utilizzate dai moduli sono in parte una implementazione simulata per l'ambiente *Unix*, come si può vedere dal codice contenuto nell'appendice A7. Il codice contenuto nell'appendice A7 (file *linux.c)* appartiene tutto alla parte di simulazione mentre negli altri file il codice appartenente al simulatore viene racchiuso tra le direttive del preprocessore: *#ifdef __SIMULATOR__* e la rispettiva chiusura *#endif*:

```
#ifdef __SIMULATOR__
kill(registeredProcess[idSource].idProcess,SIGSTOP);
#endif
//  Segna il processo come bloccato
setProcessStatus(idSource,BLOCKED);
//  Se è necessario sceglie un nuovo processo da eseguire
if (currentProcess == idSource) scheduler();
```

Buona parte del codice appartenente al simulatore verrà sostituita da procedure scritte in *ASSEMBLY* che serviranno alla comunicazione con l'hardware. Molto altro codice verrà solamente escluso poiché non necessario all'esecuzione finale.

Subito dopo l'avvio il *kernel* entra in un ciclo infinito (ciclo di *IDLE*) e verrà richiamato dall'arrivo dei segnali *SIGSCHE* e *SIGMESS*: il primo viene spedito per richiamare lo scheduler il secondo per il messager. Ogni volta che arriva una richiesta viene identificato il destinatario e creato un nuovo *thread* di gestione. I *thread* sono necessari per assicurare che durante la mutua esclusione nell'accesso alle strutture dati del *kernel* il processo non si blocchi. Supponiamo per esempio che il processo riceva un segnale *SIGMESS* di gestione di un messaggio ed abbia fatto una operazione di *down* sul semaforo che assicura la mutua esclusione. Durante l'esecuzione del messager arriva un segnale *SIGSCHE* che controlla lo stato del semaforo effettuando un'operazione *down* sul semaforo precedente e si blocca. Ma il processo era unico e non potrà più riprendere la propria esecuzione poiché attenderà un evento che solo lui può produrre. D'altra parte la mutua esclusione è necessaria. Supponiamo che sia in esecuzione il messager e che stia gestendo un messaggio che porta il processo correntemente in esecuzione in *BLOCKED*. Supponiamo che dopo l'istruzione che assegna lo stato *BLOCKED* il messager viene interrotto e lo scheduler esegue un context switch che cambia lo stato del processo in *READY*. Una volta ripreso il normale flusso del programma il messager blocca il processo. Ovviamente il risultato è quello di far continuare l'esecuzione di un processo che invece dovrebbe essere bloccato poiché lo scheduler troverà il processo *READY*. Le conseguenze possono essere più o meno disastrose e non sempre prevedibili.

Lo scheduler di *OSGM* utilizza uno schema *round robin* semplice anche se la lista utilizzata dall'algoritmo può essere ordinata in base ad una chiave di priorità stabilita in base a molti criteri differenti. Lo schedulatore in condizioni normali sarà attivato ogni n-esimo *tick* generato dal *clock* di sistema mentre in fase di simulazione viene utilizzato un intervallo di tempo molto più lungo. Nonostante il intervallo di tempo lungo i moduli che scambiano in prevalenza messaggi ritorneranno subito il controllo al sistema operativo poiché si comportano come processi *IO bound*, utilizzando solo in parte il tempo fornito. Inoltre con un intervallo molto breve ci si potrebbe trovare in una situazione molto particolare illustrata dalla figura 3.

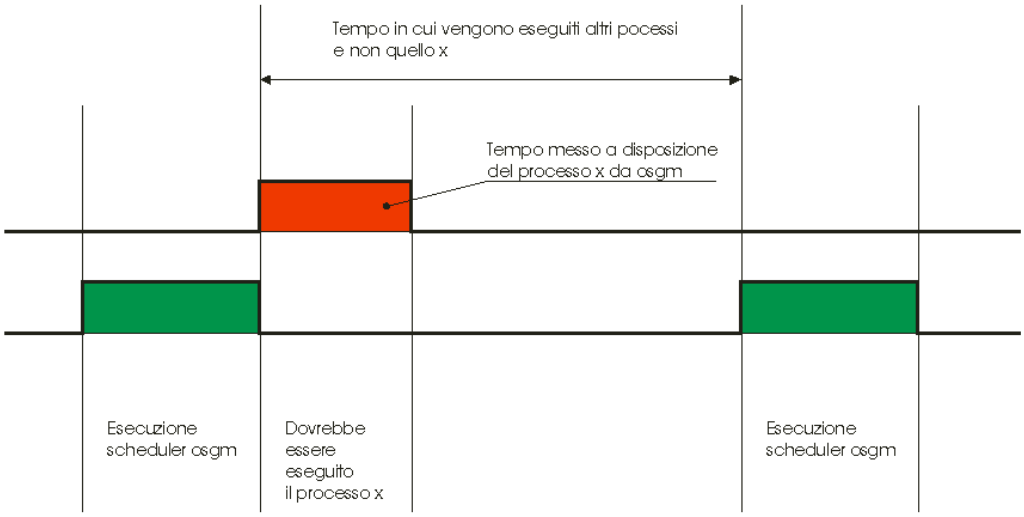

Figura 3. Rappresentazione dell'anomalia causata dalla sovrapposizione di più schedulatori.

La figura 3 illustra una situazione che può accadere realmente. Può succedere che l'intervallo fornito al processo di *OSGM* eseguito nel simulatore sia insufficiente e non venga sfruttato per l'esecuzione. Supponiamo che arrivi il segnale allo scheduler di *OSGM* che effettua un cambio di contesto a favore di un processo *x*. Subito dopo lo scheduler interno a *Unix* riprende il controllo ed esegue altri processi fin quanto non viene rigenerato il segnale di *scheduling* per *OSGM* che effettua il cambio di contesto con la conseguenza che il processo *x* non è stato ancora eseguito. La soluzione sarebbe quella di scegliere un intevallo molto grande rispetto a quello utilizzato da *Unix* e al numero di processi in esecuzione nell'ambiente.

Il *file system* è costituito essenzialmente da un *buffer* circolare a cui sia il produttore che il consumatore possono accedere effettuando chiamate di sistema. Essendo un'area di memoria condivisa si dovrebbe assicurare l'accesso esclusivo per evitare situazioni in cui il *file system* non sia più consistente. La mutua esclusione viene assicurata proprio dall'uso dei messaggi inviati dalle system call che possono essere considerate come istruzioni atomiche. A livello del simulatore l'area di memoria del *file system* è implementata attraverso l'uso di un *file* nella *directory* corrente il cui nome è *shared*. I puntatori di lettura e scrittura del *buffer* circolare (*ring*) vengono conservati in uscita dal simulatore in un *file* il cui nome è *pointers*.

3.2.3 Organizzazione della Memoria

Il programma si occupa di gestire la memoria *RAM*: allocarla e liberarla sotto richiesta dei processi. Come ogni modulo anche il gestore della memoria scambia messaggi con il resto del sistema. L'allocazione avviene utilizzando un algoritmo *first fit*, le strutture dati vengono memorizzate all'inizio di ogni blocco, mentre i blocchi liberi sono ordinati per indirizzo e riuniti in una lista *simple linked* (figura 4).

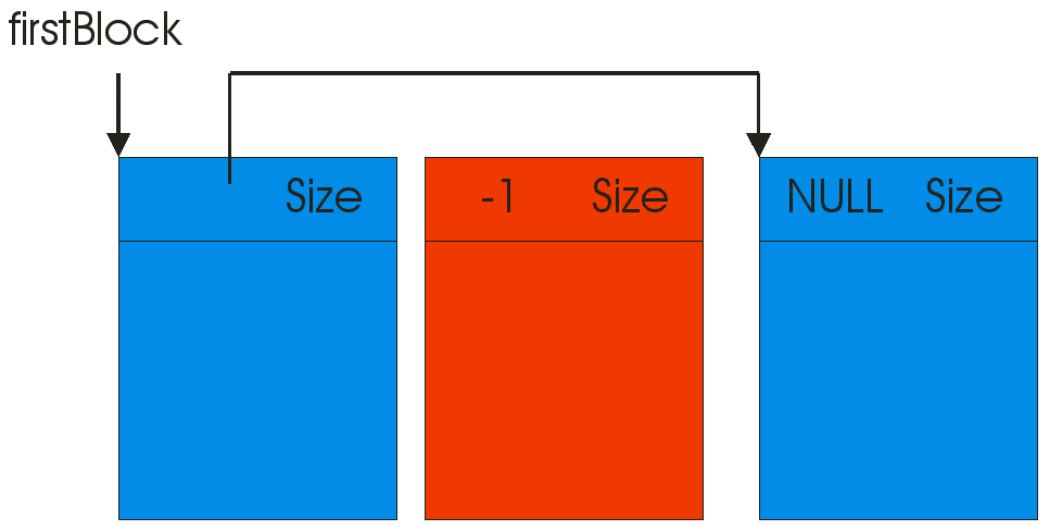

Figura 4. Rappresentazione dell'organizzazione della moria *RAM* di *OSGM* (*simpled linked list*).

Considerando che la lista può essere letta anche dalla parte grafica del simulatore bisogna assicurare la mutua esclusione attraverso un semaforo. In mancanza della mutua esclusione può verificarsi qualche spiacevole situazione. Supponiamo che il gestore della memoria stia modificando le strutture dati della *RAM*. Supponiamo che lo scheduler effettui un cambio di contesto durante tali modifiche e che un processo o lo stesso scheduler richieda un aggiornamento della schermata grafica. Cosa succede se era stato modificato un puntatore ma il blocco seguente non era stato creato? Si arriva in uno stato inconsistente. La soluzione potrebbe essere quella di specificare il tipo di resfresh da effettuare. Ma cosa succederebbe se il segnale di aggiornamento arrivasse dall'ambiente a causa del ridimensionamento della finestra del simulatore? L'effetto sarebbe comunque indeterminato. Si dovrebbe introdurre una variabile di stato per indicare la validità delle strutture dati del gestore della memoria, ma tutto sommato tra processi concorrenti una variabile di stato non può che essere implementata da un semaforo.

3.2.4 Gestione dell'input/output

Il programma si occuperà in forma definitiva di accedere alle periferiche di *IO*. Attualmente è costituito solo da una piccola porzione di codice dimostrativa ma in futuro comprenderà molto codice scritto prevalentemente in *ASSEMBLY*.

3.2.5 Gestione Magnetometro

Il programma si occupa di acquisire le misure magnetiche richiedendole al modulo di *IO* e successivamente richiedere una scrittura al *file system* (area permanente).

3.2.6 Sistema completo

Per completare il sistema operativo in futuro bisognerà soltanto aggiungere quelle parti che dipendono strettamente dall'hardware e riprendere, eventualmente adattandola, la porzione di codice già usata per *Mag-Net* scritta in *C++* e che viene eseguita attualmente dal *PC/104*.

Conclusioni

Applicare i principi della multiprogrammazione ad un sistema di acquisizione permette di ottimizzare l'uso delle risorse, permette di semplificare la programmazione a livello applicativo e l'iterazione tra i diversi compiti (*task*), ottendendo indiscutibili benefici di efficienza e stabilità. Il tutto si ottiene preservando il modello di sviluppo originariamente impiegato che permette, grazie alla separazione in due parti del sistema operativo e al simulatore, di velocizzare, automatizzare e semplificare la programmazione del sistema.

Ringraziamenti

Volevamo ringraziare ancora una volta tutta la Segreteria di Redazione del CEN che si è dimostrata molto veloce ed efficiente. Un ringraziamento particolare è dovuto, alla dott.ssa Rossella Celi per la sua cordialità, la disponibilità e la professionalità. Ringraziamo infine il Dott. Paolo Bagiacchi per l'accuratezza, la precisione e la professionalità adottata durante la revisione. Grazie.

Bibliografia

An Introduction to Project Athena, Cambridge, MA: MIT Press, 1983.

Scheifler R.W., Gettys J., (1996). *X Window System: Core and extension protocols: X version 11*, releases 6 and 6.1, Digital Press 1996, ISBN 1-55558-148-X.

Fountain A., Ferguson P., (2001). *Motif Reference Manual, Open Source Edition,* O'Reilly & Associates.

Del Negro C., Napoli R., Sicali A., (2002). *Automated system for magnetic monitoring of active volcanoes*, *Bull. Volcanol*. 64, 94-99.

Arpaci-Dusseau R.H., Arpaci-Dusseau A.C., (2014). *Operating Systems: Three Easy Pieces,* Arpaci-Dusseau Books

Sicali A., Amantia A., Cappuccio P., (2016). *Linee guida e criticità nella progettazione di sistemi per l'acquisizione di dati geofisici in prossimità di vulcani attivi.* Rapporti Tecnici INGV n°. 347, ISSN 2039-7941

ISSN 2039-7941
Anno **2018**_Numero **399**

Appendici

Appendice A1. Codice sorgente: xosgm.h

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.xosgm.h
    Created by Antonino Sicali
*/

#ifndef _XOSGM_H
#define _XOSGM_H

#include <X11/Intrinsic.h>

#define PROCESSES_TITLE        "Processes"      //  Indica il titolo della porzione di finestra grafica dedicata
                                                // ai processi
#define MEMORY_TITLE           "Memory"         //  Indica il titolo della porzione di finestra grafica dedicata
                                                // alla memoria comune
#define SHARED_MEMORY_TITLE    "Shared Memory"  //  Indica il titolo della porzione di finestra grafica dedicata
                                                // alla memoria permanente

#define KernelFileName         "kernel"         //  Indica il nome del file che contiene il codice del kernel
                                                // di osgm

#define SPACE                  10               //  Definisce lo spazione tra il nome del processo e la linea di
                                                // separazione verticale

typedef struct {                                //  Questa struttura memorizza i parametri dei processi che si
                                                // che si vogliono visualizzare
    int tick;                                   //  Contiene il numero di contextSwitch di cui il processo ha
                                                // usufruito
    char name[64];                              //  Contiene il nome del processo
    int status;                                 //  Contiene lo stato del processo
} XProcessParamShow;

//  Questa funzione si occupa di ridisegnare il contenuto della finestra grafica
void  redrawClientArea(Widget w);

//  Questa funzione si occupa di visualizzare parametri riguardanti la memoria comune a tutti i processi
// creati dal simulatore.
void redrawMemoryBox(Window win,GC gc,Dimension width,Dimension height);

//  Questa funzione si occupa di visualizzare parametri riguardanti l'esecuzione dei processi creati dal
// simulatore.
void redrawProcessesBox(Window win,GC gc,Dimension width,XFontStruct *fontInfo);

//  Questa funzione si occupa di visualizzare parametri riguardanti l'esecuzione dei processi creati dal
// simulatore.
void redrawSharedMemoryBox(Window win,GC gc,Dimension height,XFontStruct *fontInfo);

//  Questa funzione permette di intercettare l'evento che viene generato ogni qualvolta la finestra
// ha bisogno di essere ridisegnata. In particolare viene generato quando la finestra viene dimensionata
void resizeMainBox(Widget w,XtPointer client,XExposeEvent *ev);

//  Questa funzione permette di intercettare l'evento che viene generato ogni qualvolta viene premuto
// il pulsante Quit.
void  quitFunction(Widget w,XtPointer client,XtPointer ev);

//  Questa funzione si occupa di gestire la comunicazione con i processi creati dal simulatore e
// richiedere eventualmente un aggiornamento della finestra grafica.
void *threadKernel(void *arg);

#endif  _XOSGM_H
```

Appendice A2. Codice sorgente: xosgm.c

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.xosgm.c
    Created by Antonino Sicali
*/

#include "kernel.h"
#include "xosgm.h"
#include "memory.h"
#include <stdio.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <string.h>
#include <pthread.h>
#include <signal.h>

#ifdef __MOTIF__
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/DrawingA.h>
#endif __MOTIF__

#ifdef __ATHENA__
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Command.h>
#include <X11/Core.h>

#ifndef XtNuserData
#define XtNuserData "userData"
#endif

#endif __ATHENA__

Display *display;                                       //  Contiene un puntatore al display correntemente aperto
Screen *screen;                                         //  Contiene un puntatore allo screen utilizzato
int mainBoxWidth=500;                                   //  Indica la lunghezza iniziale della finestra
principale
int mainBoxHeight=300;                                  //  Indica l'altezza iniziale della finestra principale
Dimension spaceToTop=0;                                 //  Contiene la distanza dell'area grafica dal limite
                                                        // superiore della finestra grafica
XColor colorRed,colorGreen,colorBlue,colorYellow,dummy; //  Contiene alcuni descrittori di colori utilizzati

void *baseMemory=0;                                     //  Contiene un puntatore alla base della memoria comune
                                                        // tra tutti i processi creati dal simulatore
int handleCore;                                         //  Contiene il descrittore di riferimento per un
                                                       // segmento
                                                        // di memoria condivisa.
int semaphoreMemory;                                    //  Contiene l'identificativo di un semaforo che assicura
                                                        // la mutua esclusione nell'accesso alla memoria comune a
                                                        // tutti i processi creati dal simulatore.

XProcessParamShow processes[NumberOfRegisteredProcess]; //  Quest'array contiene informazioni sui processi che
                                                        // che vengono visualizzate graficamente.
long allContextSwitch=0;                                //  Contiene il numero totale di context switch
                                                            //effettuati
                                                        // dallo scheduler.
char sharedMemoryRead[128]="";                          //  Contiene informazioni dimostrative sul puntatore di
                                                        // di lettura dalla memoria permanente
char sharedMemoryWrite[128]="";                         //  Contiene informazioni dimostrative sul puntatore di
                                                        // di scrittura nella memoria permanente
Widget clientArea;                                      //  Contiene un riferimento al widget dell'area grafica
                                                        // contenuta nella finestra principale.

int main(argc,argv)
    int argc;                           //  Contiene il numero di parametri ricevuti dal processo padre
    char *argv[];                       //  Contiene un puntatore ad un array di stringhe Ascii-Z che rappresentano
                                        // i parametri di ingresso.
{
    XtAppContext applicationContext;    //  Contiene un riferimento al context dell'applicatione grafica
    Widget applicationWidget;           //  Contiene un riferimento al widget dell'applicatione grafica
    Widget mainBox;                     //  Contiene un riferimento al widget della finestra principale
    #ifdef __MOTIF__
    Widget mainMenu;                    //  Contiene un riferimento al widget della barra dei menu
    #endif __MOTIF__
    Widget quitItem;                    //  Contiene un riferimento al widget del pulsante di uscita
    pthread_t id;                       //  Contiene l'identificativo del thread che gestisce la parte grafica
    XEvent event;                       //  Questa struttura permette la gestione degli eventi ricevuti
                                        // dall'applicazione.

    //  Porzione di codice necessaria per la libreria grafica Motif
    #ifdef __MOTIF__
    //  Crea un'applicazione grafica per il simulatore
        applicationWidget=XtVaAppInitialize(&applicationContext,"osgm Xsimulator",NULL,0,&argc,argv,
                                NULL,XmNwidth,mainBoxWidth,XmNheight,mainBoxHeight,NULL);
    //  Crea la  finestra principale
```

```
        mainBox=XtVaCreateManagedWidget("mainBox",xmMainWindowWidgetClass,applicationWidget,NULL);
    //  Crea la barra dei menu
        mainMenu=(Widget) XmCreateMenuBar(mainBox,"mainMenu",NULL,0);
    //  Crea un pulsante che permette di uscire dall'applicazione
    quitItem=XtVaCreateManagedWidget("Quit",xmCascadeButtonWidgetClass,mainMenu,NULL);
    //  Crea l'area grafica su cui verranno visualizzati i dati
    clientArea = XtVaCreateWidget("clientArea",xmDrawingAreaWidgetClass,mainBox,NULL);
    //  Inizializza la barra dei menu
    XtManageChild(mainMenu);
    //  Inserisce una funzione di callback che intercetta la pressione del pulsante Quit
    XtAddCallback(quitItem,XmNactivateCallback,quitFunction,NULL);
    //  Inizializza la finestra principale collegandovi barra dei menu e area grafica
    XtVaSetValues(mainBox,XmNmenuBar,mainMenu,XmNworkWindow,clientArea,NULL);
    #endif __MOTIF__

    //  Porzione di codice necessaria per la libreria grafica Athena
    #ifdef __ATHENA__
    //  Crea un'applicazione grafica per il simulatore
    applicationWidget=XtVaAppInitialize(&applicationContext,"osgm Xsimulator",NULL,0,&argc,argv,NULL,NULL);
    //  Crea la  finestra principale
    mainBox = XtVaCreateManagedWidget("mainBox", boxWidgetClass, applicationWidget,NULL,0);
    //  Crea un pulsante che permette di uscire dall'applicazione
    quitItem = XtVaCreateManagedWidget("Quit", commandWidgetClass,mainBox,NULL);
    //  Calcola la distanza dal limite superiore
    XtVaGetValues(quitItem,XtNheight,&spaceToTop,NULL);
    spaceToTop *= 2;
    //  Crea l'area grafica su cui verranno visualizzati i dati
    clientArea = XtVaCreateManagedWidget("clientArea",coreWidgetClass,mainBox,
                                         XtNwidth,mainBoxWidth,XtNheight,mainBoxHeight,NULL);
    //  Inserisce una funzione di callback che intercetta il ridimensionamento della finestra
    // principale
    XtAddEventHandler(mainBox, ExposureMask, FALSE,(XtEventHandler) resizeMainBox, NULL);
    //  Inserisce una funzione di callback che intercetta la pressione del pulsante Quit
    XtAddCallback(quitItem,XtNcallback,quitFunction,NULL);
    #endif __ATHENA__


    display = XtDisplay(clientArea);    //  Acquisisce il display su cui si sta lavorando
    screen  = XtScreen(clientArea);     //  Acquisisce lo screen su cui si sta lavorando

    //  Crea tutti i colori di cui si avrà bisogno in seguito
    XAllocNamedColor(display,DefaultColormap(display,DefaultScreen(display)),"red",&colorRed,&dummy);
    XAllocNamedColor(display,DefaultColormap(display,DefaultScreen(display)),"green",&colorGreen,&dummy);
    XAllocNamedColor(display,DefaultColormap(display,DefaultScreen(display)),"blue",&colorBlue,&dummy);
    XAllocNamedColor(display,DefaultColormap(display,DefaultScreen(display)),"yellow",&colorYellow,&dummy);

    //  Inizializza l'area grafica della finestra principale
    XtManageChild(clientArea);
    //  Inizializza l'applicazione
    XtRealizeWidget(applicationWidget);
    //  Avvia il primo refresh sulla finestra grafica
    redrawClientArea(clientArea);
    //  Genera un thread che si occuperà di gestire l'aggiornamento dei dati visualizzati e
    // del simulatore, comunicando con tutti i processi figli.
    pthread_create(&id,NULL,threadKernel,NULL);

    while (1) {
        //  Verifica se ci sono eventi da gestire
        if (XtAppPending(applicationContext)) {
                //  Acquisisce un evento
                XtAppNextEvent(applicationContext, &event);
                //  Verifica se l'evento richiede l'aggiornamento della finestra grafica
                if (event.type == Expose && event.xexpose.window == XtWindow(clientArea))
                        redrawClientArea(clientArea);
                //  Inoltra ai singoli gestori gli eventi letti
                XtDispatchEvent(&event);
                                        }
         }
}

//  Questa funzione si occupa di ridisegnare il contenuto della finestra grafica
void redrawClientArea(w)
    Widget w;   //  Indica il widget associato alla finestra grafica utilizzata per visualizzare
                // i dati relativi all'esecuzione del simulatore.
{
    int x;
    Drawable win;
    Dimension width,height;                         //  Contengono la lunghezza e l'altezza della
                                                    // finestra grafica
    int fontHeight;                                 //  Contiene l'altezza del font utilizzato
    XFontStruct *fontInfo;                          //  Contiene informazioni relative al font utilizzato
    GC gc;                                          //  Contiene il Graphics Context associato alla
                                                    // finestra grafica

    win = XtWindow(w);                              //  Acquisisce un riferimento alla finestra grafica

    gc = XCreateGC(display, win, 0, NULL);          //  Crea un Graphics Context per disegnare sulla
                                                    // finestra grafica.
    fontInfo=XLoadQueryFont(display,"fixed");       //  Acquisisce informazioni sul font utilizzato
```

```
    fontHeight = fontInfo->ascent+fontInfo->descent;    //  Calcola l'altezza del font utilizzato

    #ifdef __MOTIF__
    XtVaGetValues(w,XmNwidth,&width,NULL);              //  Acquisisce la lunghezza della finestra grafica
    XtVaGetValues(w,XmNheight,&height,NULL);            //  Acquisisce l'altezza della finestra grafica
    #endif __MOTIF__

    #ifdef __ATHENA__
    XtVaGetValues(w,XtNwidth,&width,NULL);              //  Acquisisce la lunghezza della finestra grafica
    XtVaGetValues(w,XtNheight,&height,NULL);            //  Acquisisce l'altezza della finestra grafica
    #endif __ATHENA__


    XClearWindow(display,win);                          //  Pulisce l'area grafica della finestra

    redrawMemoryBox(win,gc,width,height);               //  Visualizza i dati relativi alla memoria comune
    redrawSharedMemoryBox(win,gc,height,fontInfo);      //  Visualizza i dati relativi alla memoria permanente
    redrawProcessesBox(win,gc,width,fontInfo);          //  Visualizza i dati relativi all'esecuzione dei
                                                        // processi.


    XSetForeground(display,gc,BlackPixelOfScreen(screen));              //  Selezione il colore nero
    XDrawLine(display,win,gc,(width*4)/5,spaceToTop,(width*4)/5,height);  //  Disegna una linea verticale di
                                                                        // separazione sulla destra
    XDrawLine(display,win,gc,0,(height*4)/5,(width*4)/5,(height*4)/5);    //  Disegna una linea orizzonatale di
                                                                        // separazione in basso

    XDrawLine(display,win,gc,SPACE,fontHeight+spaceToTop+SPACE,         //  Disegna una linea orizzonatale di
            (width*4)/5-SPACE,fontHeight+spaceToTop+SPACE);             // separazione in alto

    //  Calcola la posizione in cui posizionare la stringa in modo che sia centrata
    x=(width*4)/10-XTextWidth(fontInfo,PROCESSES_TITLE,strlen(PROCESSES_TITLE))/2;
    //  Visualizza il testo desiderato
    XDrawString(display,win,gc,x,fontHeight+spaceToTop,PROCESSES_TITLE,strlen(PROCESSES_TITLE));
    //  Calcola la posizione in cui posizionare la stringa in modo che sia centrata
    x=(width*4)/5+width/10-XTextWidth(fontInfo,MEMORY_TITLE,strlen(MEMORY_TITLE))/2;
    //  Visualizza il testo desiderato
    XDrawString(display,win,gc,x,fontHeight+spaceToTop,MEMORY_TITLE,strlen(MEMORY_TITLE));
    //  Calcola la posizione in cui posizionare la stringa in modo che sia centrata
    x=(width*4)/10-XTextWidth(fontInfo,SHARED_MEMORY_TITLE,strlen(SHARED_MEMORY_TITLE))/2;
    //  Visualizza il testo desiderato
    XDrawString(display,win,gc,x,(height*4)/5+fontHeight,SHARED_MEMORY_TITLE,strlen(SHARED_MEMORY_TITLE));
    //  Libera di Graphics Context poichè non è più necessario
    XFreeGC(display, gc);
}

//  Questa funzione si occupa di visualizzare parametri riguardanti la memoria comune a tutti i processi
// creati dal simulatore.
void redrawMemoryBox(win,gc,width,height)
    Window win;          //  Contiene il descrittore della finestra grafica su cui si vogliono visualizzare
                         // le informazioni
    GC gc;               //  Contiene il descrittore del Graphics Context utilizzato per disegnare sulla
                         // finestra grafica
    Dimension width;     //  Contiene la lunghezza della finestra grafica
    Dimension height;    //  Contiene l'altezza della finestra grafica
{
    double dy;           //  Conterrà il numero di pixel associato ad ogni byte della memoria comune
    double y=spaceToTop; //  Conterrà in ogni instante l'ordinata a cui inizierà ad essere disegnato il
                         // blocco corrente.
    Node *this;          //  Questa struttura è necessaria per accedere alle informazioni presenti ad
                         // inizio blocco.

    if (!baseMemory) return;   //  Non è stata ancora allocata la memoria comune
    down(semaphoreMemory);     //  Si riserva l'accesso esclusivo alla memoria comune
    dy=height;                 //  Calcola il numero di pixel associato ad ogni byte della memoria comune
    dy /= MemorySize;
    this=(Node *) baseMemory;  //  Inizializza il puntatore al primo blocco


    do {
    //  Sceglie il colore con cui il blocco verrà visualizzato: rosso se il blocco è occupato
    // blue se il blocco è libero
    if (this->p == (char *) -1) XSetForeground(display,gc,colorRed.pixel);
                           else
                           XSetForeground(display,gc,colorBlue.pixel);
    //  Disegna un rettangolo pieno che rappresenta il blocco di memoria
    XFillRectangle(display,win,gc,(width*4)/5,y,width,y+this->size*dy);
    //  Si sposta al blocco successivo sia come posizione grafica sullo schermo sia come posizione in
    // memoria comune.
    y += this->size*dy;
    this = (Node *) (((char *) this)+this->size);
    //  Verifica che ci siano ancora blocchi da disegnare
    } while(((void *) this) < baseMemory+MemorySize);
    //  Rilascia la memoria comune
    up(semaphoreMemory);
}
```

```
      //  Questa funzione si occupa di visualizzare parametri riguardanti l'esecuzione dei processi creati dal
// simulatore.
void redrawProcessesBox(win,gc,width,fontInfo)
    Window win;             //  Contiene il descrittore della finestra grafica su cui si vogliono visualizzare
                            // le informazioni
    GC gc;                  //  Contiene il descrittore del Graphics Context utilizzato per disegnare sulla
                            // finestra grafica
    Dimension width;        //  Contiene la lunghezza della finestra grafica
    XFontStruct *fontInfo;  //  Contiene informazioni relative al font utilizzato
{
    double y;               //  Contiene la posizione dove visualizzare le caratterisctiche del
                            // processo selezionato
    char status[10];        //  Buffer si supporto per trasformare lo stato di un processo in modalità
                            // testo.

    //  Calcola l'altezza del font utilizzato
    int fontHeight=fontInfo->ascent+fontInfo->descent;
    int i=0,x,dx;

    //  Calcola la posizione dove visualizzare le caratterisctiche del primo processo
    y = fontHeight*2+spaceToTop+SPACE;

    //  Scandisce tutti gli elementi dell'array per rintracciare quali processi sono in esecuzione
    for (i=0;i<NumberOfRegisteredProcess;i++)
        //  Se un elemento dell'array rappresenta un processo il numero di contextSwitch (tick) è
        // diverso da zero.
        if (processes[i].tick) {
                    //  Calcola la posizione per un allineamento del nome del processo a destra
                    x=width/5-XTextWidth(fontInfo,processes[i].name,strlen(processes[i].name))-SPACE;
                    //  Seleziona il colore nero
                    XSetForeground(display,gc,BlackPixelOfScreen(screen));
                    //  Visualizza il nome del processo
                    XDrawString(display, win, gc,x,y,processes[i].name,strlen(processes[i].name));
                    //  Disegna una linea orizzonatale di separazione verticale
                    XDrawLine(display,win,gc,width/5,y,width/5,y-fontHeight);
                    //  Trasforma lo stato del processo in una stringa Ascii-Z e sceglie
                    // un colore opportuno.
                    switch(processes[i].status) {
                                    case RUNNING:
                                        XSetForeground(display,gc,colorGreen.pixel);
                                        strcpy(status,"(Running)");
                                        break;
                                    case READY:
                                        XSetForeground(display,gc,colorYellow.pixel);
                                        strcpy(status,"( Ready )");
                                        break;
                                    case BLOCKED:
                                        XSetForeground(display,gc,colorRed.pixel);
                                        strcpy(status,"(Blocked)");
                                        break;
                                                }
                    //  Calcola la posizione per un allineamento dello stato del processo a sinistra
                    x=XTextWidth(fontInfo,status,strlen(status))+3*SPACE;
                    //  Visualizza lo stato del processo
                    XDrawString(display, win, gc,width/5+SPACE,y,status,strlen(status));
                    //  Seleziona il colore nero
                    XSetForeground(display,gc,BlackPixelOfScreen(screen));
                    //  Disegna una linea orizzonatale di separazione verticale
                    XDrawLine(display,win,gc,width/5+x-SPACE,y,width/5+x-SPACE,y-fontHeight);
                    //  Seleziona il colore verde
                    XSetForeground(display,gc,colorGreen.pixel);
                    //  Disegna un rettangolo la cui lunghezza è direttamente proporzionale al
                    // numero di contextSwitch ricevuti
                    dx=(((width*3)/5-x-2*SPACE)*processes[i].tick)/allContextSwitch;
                    XFillRectangle(display,win,gc,width/5+x,y-fontHeight/2,dx,fontHeight/2);
                    XSetForeground(display,gc,colorYellow.pixel);
                    //  Disegna un rettangolo la cui lunghezza è direttamente proporzionale al
                    // numero di contextSwitch ricevuti
                    XFillRectangle(display,win,gc,width/5+x+dx,y-fontHeight/2,
                                    (width*3)/5-x-2*SPACE-dx,
                                    fontHeight/2);
                    //  Calcola la posizione in cui verranno visualizzate le caratteristiche del
                    // prossimo processo
                    y += fontHeight;
                              }
}

//  Questa funzione si occupa di visualizzare parametri riguardanti l'esecuzione dei processi creati dal
// simulatore.
void redrawSharedMemoryBox(win,gc,height,fontInfo)
    Window win;             //  Contiene il descrittore della finestra grafica su cui si vogliono visualizzare
                            // le informazioni
    GC gc;                  //  Contiene il descrittore del Graphics Context utilizzato per disegnare sulla
                            // finestra grafica
    Dimension height;       //  Contiene l'altezza della finestra grafica
    XFontStruct *fontInfo;  //  Contiene informazioni relative al font utilizzato
{
    //  Calcola l'altezza del font utilizzato
    int fontHeight=fontInfo->ascent+fontInfo->descent;
```

```
    //  Seleziona il colore verde
    XSetForeground(display,gc,BlackPixelOfScreen(screen));
    //  Visualizza le informazioni relative al puntatore di lettura dalla memoria comune
    XDrawString(display, win, gc,10,(height*4)/5+fontHeight*2,sharedMemoryRead,strlen(sharedMemoryRead));
    //  Visualizza le informazioni relative al puntatore di scrittura nella memoria comune
    XDrawString(display, win, gc,10,(height*4)/5+fontHeight*3,sharedMemoryWrite,strlen(sharedMemoryWrite));
}

#ifdef __ATHENA__
//  Questa funzione permette di intercettare l'evento che viene generato ogni qualvolta la finestra
// ha bisogno di essere ridisegnata. In particolare viene generato quando la finestra viene dimensionata
void resizeMainBox(w,client,ev)
    Widget w;               //  Contiene un riferimento al widget che ha generato l'evento GraphicsExpose
    XtPointer client;       //  Contiene informazioni aggiuntive passate alla callback (NULL)
    XExposeEvent *ev;       //  Contiene informazioni relative all'evento generato (GraphicsExpose)
{
    Dimension width,height; //  Memorizza temporaneamente la lunghezza e l'altezza della finestra principale

    XtVaGetValues(w,XtNwidth,&width,NULL);            //  Acquisisce la lunghezza della finestra principale
    XtVaGetValues(w,XtNheight,&height,NULL);          //  Acquisisce l'altezza della finestra principale
    XtConfigureWidget(clientArea,0,0,width,height,0); //  Cambia le dimensioni della finestra grafica
}
#endif __ATHENA__

//  Questa funzione permette di intercettare l'evento che viene generato ogni qualvolta viene premuto
// il pulsante Quit.
void quitFunction(w,client,ev)
    Widget w;               //  Contiene un riferimento al widget che ha generato l'evento
    XtPointer client;       //  Contiene informazioni aggiuntive passate alla callback (NULL)
    XtPointer ev;           //  Contiene informazioni relative all'evento generato
{
    exit(0);
}

//  Questa funzione si occupa di gestire la comunicazione con i processi creati dal simulatore e
// richiedere eventualmente un aggiornamento della finestra grafica.
void *threadKernel(arg)
    void *arg;          //  Argomenti passati al thread dalla funzione pthread_create()
{
    int tmp[2];         //  Contiene un nodo di pipe per la comunicazione tra la parte grafica ed i
                        // processi creati dal simulatore
    char args[5][16];   //  E' un array in vengono memorizzati i parametri da passare al kernel di osgm
    char *argv[]={args[0],args[1],args[2],args[3],args[4],NULL};
    MessageXSimulator m; //  Questa struttura è necessaria per leggere nel modo corretto le
                        // notifiche ricevute
    char **p;
    int   *q;

    //  Inizializza l'array che contiene le informazioni dei processi
    memset(processes,0,sizeof(XProcessParamShow)*NumberOfRegisteredProcess);

    pipe(tmp);                                              //  Crea un nodo di pipe
    handleCore=shmget(0,MemorySize,S_IRWXU | IPC_CREAT);    //  Crea un segmento di memoria condivisa e lo
    baseMemory=shmat(handleCore,0,0);                       // lega ad un indirizzo di memoria
    initMemory(baseMemory);                                 //  Inizializza l'area di memoria comune ai
                                                            // processi, creando un unico blocco libero.
    semaphoreMemory=semget(0,1,S_IRWXU | IPC_CREAT);        //  Crea un semaforo per l'accesso in mutua
                                                            // esclusione alla memoria comune.
    up(semaphoreMemory);                                    //  Inizializza il semaforo
    strcpy(args[0],KernelFileName);                         //  Inizializza l'array dei parametri da
    sprintf(args[1],"%d",tmp[1]);                           // passare al kernel di osgm.
    sprintf(args[2],"%d",handleCore);
    sprintf(args[3],"%p",baseMemory);
    sprintf(args[4],"%d",semaphoreMemory);

    if (!fork()) execv(KernelFileName,argv);                //  Crea un nuovo processo in Unix (Linux)
                                                            // e successivamente richiama dal disco il
                                                            // kernel di osgm.
    while(1) {
        read(tmp[0],&m,sizeof(MessageXSimulator));          //  Ascolta la pipe per rilevare notifiche
                                                            // dai processi.
        switch(m.id) {
                //  Identica un'avvenuta modifica delle strutture dati del kernel
                case ID_PROCESS_REFRESH:
                    q=(int *)(m.body);
                    //  Il puntatore q punta allo stato del processo. q+1 punta all'identificatore
                    // del processo. q+2 punta al nome del processo
                    if (*q == RUNNING) { processes[*(q+1)].tick++; allContextSwitch++; }
                    //  Acquisisce lo stato del processo
                    processes[*(q+1)].status=*q;
                    //  Acquisisce il nome del processo
                    strcpy(processes[*(q+1)].name,(char *) (q+2));
                    break;
                //  Identica un'avvenuta modifica delle strutture dati del gestore della memoria comune
                case ID_MEMORY_REFRESH:
                    break;
                //  Identica un'avvenuta modifica del puntatore di lettura dalla memoria permanente
                case ID_READ_SHARED_MEMORY_REFRESH:
```

```
                    //  Acquisisce le informazioni da visualizzare relative al puntatore di lettura
                    // dalla memoria permanente
                    strcpy(sharedMemoryRead,m.body);
                    break;
                //  Identica un'avvenuta modifica del puntatore di scrittura nella memoria permanente
                case ID_WRITE_SHARED_MEMORY_REFRESH:
                    //  Acquisisce le informazioni da visualizzare relative al puntatore di scrittura
                    // nella memoria permanente
                    strcpy(sharedMemoryWrite,m.body);
                    break;
                     }
        redrawClientArea(clientArea);                   //  Ridisegna l'area grafica
         }
}
```

Appendice A3. Codice sorgente: kernel.h

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.kernel.h
    Created by Antonino Sicali
*/

#ifndef _KERNEL_H
#define _KERNEL_H

#include "osgm.h"

#define SIGMESS                     SIGUSR1                 //  Indica il segnale di notifica per il messager
#define SIGSCHE                     SIGALRM                 //  Indica il segnale di notifica per lo scheduler
#define SIGTRIG                     SIGUSR2                 //  Indica il segnale di notifica per il trigger

#define NumberOfRegisteredProcess   32                      //  Indica il numero massimo di processi creabili
#define ALLPROCESSES                NumberOfRegisteredProcess //  Indica l'identificativo da utilizzare per
                                                            // spedire o ricevere messaggi da tutti i processi
#define IDKERNEL                    NumberOfRegisteredProcess //  Indica l'identificativo del kernel di osgm

typedef struct {                            //  Questa struttura rappresenta un elemento della tabella dei
    #ifdef __SIMULATOR__                    // processi
    int idProcess;                          //  Contiene il pid del processo Unix (Linux) che contiene il
                                            // processo osgm
    int pipeToProcess;                      //  Contiene il descrittore della pipe utilizzata per comunicare
                                            // con il processo
    char name[64];                          //  Contiene il nome del processo
    #endif __SIMULATOR__
    int used;                               //  Indica se l'elemento è utilizzato
    int status;                             //  Contiene lo stato corrente del processo
    int next;                               //  Contiene un puntatore al prossimo elemento nella lista dello
                                            // scheduler
    int prev;                               //  Contiene un puntatore al elemento precedente nella lista dello
                                            // scheduler
    int priority;                           //  Indica la priorità posseduta dal processo
    int decodeWait;                         //  Indica che il processo attende che un messaggio gli venga
                                            // decodificato
    char undecodeMessage[MessageLength]; //  Contiene il nome del messaggio da decodificare
} Process;

typedef struct {                            //  Questa struttura rappresenta un elemento della tabella dei
                                            // messaggi
    int idProcess;                          //  Specifica l'identificativo del processo che ha registrato il
                                            // messaggio
    char name[MessageLength];               //  Contiene il nome del messaggio
} Message;

#define SharedMemoryGranularity     30  //  Indica lo spostamento compiuto dalla testina di lettura se
                                        // viene a sovrapporsi a quella di scrittura
#define SHARED_MEMORY_LENGTH 30000      //  Indica quanti byte è lunga la memoria permanente
#ifdef __SIMULATOR__
#define SharedMemoryPointers "pointers" //  Indica quale è il file utilizzato per conservare i puntatori
                                        // all'area di memoria permanente
#define SharedMemoryFile     "shared"   //  Indica quale è il file utilizzato per implementare la memoria
                                        // permanente
#endif __SIMULATOR__

typedef struct {                        //  Questa struttura viene utilizzata per richiedere una lettura o
                                        // scrittura nell'area permanente
    char *p;                            //  Contiene un puntatore al buffer di lettura/ scrittura
    int n;                              //  Specifica quanti byte bisogna leggere o scrivere
} SharedData;

//  Questo messaggio viene spedito al kernel quando si vuole scrivere nella memoria permanente (FileSytem)
#define WRITE_SHARED_MEMORY         NumberOfRegisteredMessage
//  Questo messaggio viene spedito al kernel quando si vuole leggere dalla memoria permanente (FileSytem)
#define READ_SHARED_MEMORY          NumberOfRegisteredMessage+1
//  Questo messaggio viene spedito al kernel quando si vuole registrare un messaggio
#define KERNEL_REGISTER_MESSAGE     NumberOfRegisteredMessage+2
//  Questo messaggio viene spedito al kernel quando si vuole decodificare un messaggio
#define KERNEL_DECODE_MESSAGE       NumberOfRegisteredMessage+3
//  Questo messaggio viene spedito ad un processo come risposta ad un messaggio
#define KERNEL_ACK                  NumberOfRegisteredMessage+4
//  Questo messaggio viene spedito al kernel quando si vuole fermare il processo
#define KERNEL_STOP_PROCESS_MESSAGE NumberOfRegisteredMessage+5
//  Questo messaggio viene spedito al kernel quando si vuole avvertire il kernel che si sta attendendo
// un messaggio
#define KERNEL_RECEIVE_MESSAGE      NumberOfRegisteredMessage+6

#define READY           0       //  Indica che il processo è in attesa per essere eseguito
#define RUNNING         1       //  Indica che il processo è in esecuzione
#define BLOCKED         2       //  Indcia che il processo è bloccato

#ifdef __SIMULATOR__
#define QUANTUM         1       //  Specifica quale il lasso di tempo massimo offerto ad un processo per la
                                // propria esecuazione. Viene espresso in secondi
#else  __SIMULATOR__
#define QUANTUM
```

```
#endif __SIMULATOR__

#define InitialPriority 0         //  Indica quale è la priorità da assegnare ad un processo appena creato

#ifdef __SIMULATOR__
#define ProcessList     "modules" //  Indica il file che contiene i nomi dei processi da avviare
#endif __SIMULATOR__

//  Questa variabile contiene il descrittore della pipe che permette di leggere i messaggi dai processi.
extern int pipeFromProcesses;

//  Contiene il descrittore associato con il segmento condiviso tra tutti i processi del
// simulatore.
extern int handleCore;

//  Contiene l'indirizzo base della memoria comune a tutti i processi
extern void *baseMemory;

//  Contiene l'identificativo del semaforo che assicura la mutua esclusione sulla
// memoria condivisa.
extern int semaphoreMemory;

//  Quest'array di strutture rappresenta la tebella dei processi
extern Process registeredProcess[NumberOfRegisteredProcess];

//  Questa funzione permette di stabilire se è possibile creare nuovi processi
int isProcessTableFull(void);

//  Questa funzione permette di registrare un processo in osgm
int registerProcess(void);

//  Questa funzione permette di eliminare un processo dal sistema
int unregisterProcess(int id);

//  Questa funzione inizializza lo stack degli elementi liberi nella tabella dei processi
void initProcessStack(void);

#ifdef __SIMULATOR__
#define DEMOSTRATION_BLOCK  30
// Questa funzione permette di eseguire un processo figlio
int startProcess(int pipeToKernel,char *filename);

//  Questa funzione permette di leggere dal file utilizzato per implementare la memoria
// permanente un blocco di caratteri per essere visualizzati dalla parte grafica
void readSharedBlock(long seek,char *buffer);

//  Questa funzione notifica alla parte grafica del simulatore che ci sono stati cambiamenti
// nella gestione della meomoria permanente.
void setSharedStatus(void);

// Questa funzione memorizza i puntatori alla memoria permanente in un file sul disco
void writePointers(void);
#endif __SIMULATOR__

#ifdef __KERNEL__
void sendMessage(int pipeToKernel,int message,void *body,int n,int idSource);
void receiveMessage(int pipeFromKernel,int *message,void *body,int *n,int *idSource);
#endif __KERNEL__

//  Questa funzione permette di effettuare un'operazione di down su un semaforo
void down(int sem);

//  Questa funzione permette di effettuare un'operazione di up su un semaforo
void up(int sem);

// Questa funzione esplica tutte le funzionalità del messager
void messager(void);

// Questa funzione esplica tutte le funzionalità dello scheduler
void scheduler(void);

//  Questa funzione si occupa di riprendere l'esecuzione di processo precedentemente interrotto
void resumeProcess(int id);

// Inserisce il processo nella lista di schedulazione
void insertProcess(int id);

// Elimina il processo dalla lista di schedulazione
void deleteProcess(int id);

//  Questa funzione si occupa di smistare le richieste inoltrate al kernel
void kernel(int signum);

//  Questa funzione si occupa di intercettare i segnali di trigger per riattivare il timer
// che chiamerà successivamente lo scheduler
void retrigger(int signum);

// Questa funzione intercetta il segnale di interruzione
void controlC(int signum);
```

```
//  Funzione principale del thread addetto allo scheduling
void *threadScheduler(void *arg);

//  Funzione principale del thread addetto al messaging
void *threadMessager(void *arg);

//  Questa funzione cambia lo stato di un processo e notifica il cambiamento alla parte grafica
// del simulatore
void setProcessStatus(int id,int status);

#endif  _KERNEL_H
```

Appendice A4. Codice sorgente: kernel.c

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.kernel.c
    Created by Antonino Sicali
*/

#include "kernel.h"

#ifdef __SIMULATOR__
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <pthread.h>
#endif __SIMULATOR__

Message registeredMessage[NumberOfRegisteredMessage]; //  Quest'array di strutture memorizza i messaggi registrati
int     registeredMessagePointer=0;                   // nel sistema. La variabile registeredMessagePointer
                                                      // memorizza il numero di messaggi registrati nel sistema.
int     undecodeMessages=0;                           //  Contiene il numero di messaggi di cui è stata richiesta
                                                      // la decodifica ma che non sono ancora registrati.
Process registeredProcess[NumberOfRegisteredProcess]; //  Quest'array di strutture rappresenta la tebella dei
int     processStack[NumberOfRegisteredProcess];      // processi. L'array processStack permette di inserire
int     processStackPointer;                          // un processo nella tabella dei processi in tempo costante.
                                                      //  La variabile processStackPointer punta alla base
                                                      // dello stack che contiene tutti gli slot liberi della
                                                      // tabella dei processi.
int     schedulerListHead=-1;                         //  Questa variabile punta all'elemento della tabella dei
                                                      // processi che dovrà essere eseguito successivamente.
int     currentProcess=-1;                            //  Questa variabile contiene il pid del processo
                                                         //attualmente in esecuzione.
long    sharedMemoryRead=0;                           //  Questa variabile punta al primo byte dell'area
                                                      // permanente da leggere;
long    sharedMemoryWrite=0;                          //  Questa variabile punta al primo byte dell'area
                                                      // permanente da scrivere;
#ifdef __SIMULATOR__
int     semaphoreKernel;                              //  Questa variabile contiene l'identificavivo del semaforo
                                                      // che assicura la mutua esclusione tra messager e scheduler
                                                      // nell'accesso alle strutture dati del kernel.
int     kernelPid;                                    //  Questa variabile contiene il pid del kernel, necessario
                                                      // per inviare il segnale di allarme dai thread del messager
                                                      // o scheduler.
int     quit=1;                                       //  Questa variabile permette di uscire dal ciclo while
                                                      // principale del kernel
int     testStop;
int     pipeFromProcesses;                            //  Questa variabile contiene il descrittore della pipe che
                                                      // permette di leggere i messaggi dai processi.
int     pipeToXSimulator=0;                           //  Questa variabile contiene il descrittore della pipe che
                                                      // permette di notificare alla parte grafica che sono stati
                                                      // cambiati parametri.
#endif __SIMULATOR__

// argv[1] contiene il descrittore della pipe di notifica alla parte grafica del simulatore
// argv[2] contiene il descrittore associato al segmento codiviso tra tutti i processi del simulatore
// argv[3] contiene un puntatore alla base della memoria codivisa tra tutti i processi del simulatore
// argv[4] contiene l'identificativo del semaforo che assicura la mutua esclusione nell'accesso alla memoria
//         condivisa
int main(argc,argv)
    int argc;           //  Contiene il numero di parametri ricevuti dal processo padre
    char *argv[];       //  Contiene un puntatore ad un array di stringhe Ascii-Z che rappresentano
                        // i parametri di ingresso.
{
    #ifdef __SIMULATOR__
    int pipeKernel[2]; //  Contiene un nodo di pipe con cui il kernel può comunicare con i processi
    FILE *handle;       //  Contiene il descrittore del file contenente tutti i processi da richiamare
                        // dal disco.
    int h;              //  Contiene il descrittore del file contenente i puntatori alla memoria permanente
    char buffer[128];   //  Buffer di lettura dei nomi dei processi dal file presente sul disco
    //  Questa struttura conterrà informazioni che permetterà la notifica alla parte grafica dei
    // cambiamenti avvenuti.
    MessageXSimulator messageXSimulator;


    //  Permette di scrivere immediatamente sullo standard di output
    setvbuf(stdout,0,_IONBF,0);
    #endif __SIMULATOR__

    initProcessStack();

    #ifdef __SIMULATOR__
    //  Verifica se esiste una parte grafica che ha passato al kernel cinque
    // parametri di inizializzazione. In caso contrario il kernel inizializza i parametri
    // necessari.
    if (argc == 5) {
```

```
        //  Converte i parametri testuali in formato binario
        sscanf(argv[1],"%d",&pipeToXSimulator);
        sscanf(argv[2],"%d",&handleCore);
        sscanf(argv[3],"%p",&baseMemory);
        sscanf(argv[4],"%d",&semaphoreMemory);
           }
           else
           {
        //  Richiede memoria condivisa al sistema operativo (Unix, Linux, etc.)
        handleCore=shmget(0,MemorySize,S_IRWXU | IPC_CREAT);
        //  Crea un semaforo per assicurare la mutua esclusione sulla memoria allocata
        semaphoreMemory=semget(0,1,S_IRWXU | IPC_CREAT);
        //  Inizializza il semaforo creato
        up(semaphoreMemory);

           }
//  Se il puntatore baseMemory è già stato inizializzato lega semplicemente il segmento
// condiviso al puntatore altrimenti inizializza il puntatore con un nuovo indirizzo.
if (!baseMemory) baseMemory=(char *) shmat(handleCore,baseMemory,0);
             else
             shmat(handleCore,baseMemory,0);
//  Verifica che non ci siano stati errori nell'allocazione della memoria condivisa
if ((int) baseMemory == -1) return -1;
//  Acquisisce il pid del kernel per un utilizzo futuro.
kernelPid=getpid();
//  Apre il file dei puntatori all'area permanente
h=open(SharedMemoryPointers,O_RDONLY);
//  Nel caso in cui il file esiste vengono letti i puntatori
if (h != -1) {
    read(h,&sharedMemoryRead,sizeof(long));
    read(h,&sharedMemoryWrite,sizeof(long));
    close(h);
         }
//  Notifica alla parte grafica che il puntatore di lettura della memoria permanente è stato modificato
messageXSimulator.id=ID_READ_SHARED_MEMORY_REFRESH;
sprintf(messageXSimulator.body,"addr(read)  0x%6.6lx: ",sharedMemoryRead);
//  Legge un blocco dimostrativo dall'area permanente
readSharedBlock(sharedMemoryRead,messageXSimulator.body+strlen(messageXSimulator.body));
if (pipeToXSimulator) write(pipeToXSimulator,&messageXSimulator,sizeof(MessageXSimulator));
//  Notifica alla parte grafica che il puntatore di scrittura della memoria permanente è stato modificato
messageXSimulator.id=ID_WRITE_SHARED_MEMORY_REFRESH;
sprintf(messageXSimulator.body,"addr(write) 0x%6.6lx: ",sharedMemoryWrite);
//  Legge un blocco dimostrativo dall'area permanente
readSharedBlock(sharedMemoryWrite,messageXSimulator.body+strlen(messageXSimulator.body));
if (pipeToXSimulator) write(pipeToXSimulator,&messageXSimulator,sizeof(MessageXSimulator));

//  Crea un nodo pipe per la comunicazione tra kernel e processi
if (pipe(pipeKernel)) { perror("osgm kernel"); return -1; }
//  Conserva in una variabile globale il descrittore di lettura dal nodo di pipe
pipeFromProcesses=pipeKernel[0];
//  Crea un array di due semafori: il primo elemento assicura la mutua esclusione nell'accesso
// alle strutture dati del kernel; il secondo implementa una variabile condivisa.
semaphoreKernel=semget(0,2,S_IRWXU | IPC_CREAT);
//  Apre in lettura il file di testo che contiene la lista di tutti i processi da eseguire
handle=fopen(ProcessList,"rt");
//  Esce se ci sono errori
if (handle == NULL) { perror("osgm kernel"); return -1;

printf("\nOSGM processes");
printf("\n\nName           ID\t");
printf("\n-------------------------------------------------------");
printf("\nKernel         %d",IDKERNEL);
//  Legge dal file ciascuna riga
while(fscanf(handle,"%s",buffer) == 1)
    if (strlen(buffer))
    if (startProcess(pipeKernel[1],buffer)) { perror("osgm kernel"); return -1;

//  Chiude il file
fclose(handle);
printf("\n");
//  Inizializza il semaforo che assicura la mutua esclusione nell'accesso alle strutture
// del kernel
up(semaphoreKernel);
signal(SIGSCHE,kernel);    //  Installa il gestore dello scheduler
signal(SIGMESS,kernel);    //  Installa il gestore del messager
signal(SIGINT,controlC);   //  Installa il gestore dell'interruzione
signal(SIGTRIG,retrigger); //  Installa il gestore del trigger
retrigger(0);              //  Avvia il trigger
#endif __SIMULATOR__

// Avvia il primo processo
currentProcess=schedulerListHead;
resumeProcess(currentProcess);

#ifdef __SIMULATOR__
while(quit);               //  Mantiene in esecuzione il kernel finchè non arriva un segnale
                           // interruzione.
writePointers();           //  Scrive i puntatori in un file sul disco
shmdt(baseMemory);         //  Libera la condivisa
```

```
    #endif __SIMULATOR__
    return 0;
}

#ifdef __SIMULATOR__
// Questa funzione intercetta il segnale di interruzione
void controlC(signum)
    int signum;                //  Numero del segnale intercettato (SIGINT)
{
    quit=0;                    //  Nega la guardia del ciclo while principale nella funzione main
}

// Questa funzione intercetta il segnale di stop inviato dai processi figli al kernel
void stopHandle(signum)
    int signum;            //  Numero del segnale intercettato (SIGCHLD)
{
    testStop=0;            //  Nega la guardia del ciclo while nella funzione startProcess
}

// Questa funzione permette di eseguire un processo figlio
int startProcess(pipeToKernel,filename)
    int pipeToKernel;      //  Contiene il descrittore della pipe utilizzata per comunicare con il kernel
    char *filename;        //  Contiene il nome del file che contiene il programma da eseguire
{
    int id;                //  Contiene il pid interno ad Unix (Linux) del processo creato
    int idProcess;         //  Contiene l'identificatore del processo interno ad osgm
    int pipeFromKernel[2]; //  Contiene un nodo di pipe che permette di inviare messaggi dal kernel
                           // ai processi.
    void (*prev)(int);     //  Memorizza temporaneamente il vecchio gestore del segnale SICHLD

    if (isProcessTableFull()) return -1;  //  Controlla se la tabella dei processi è piena
    pipe(pipeFromKernel);                 //  Crea un nodo di pipe per la comunicazione tra kernel
                                          // e processo.
    idProcess=registerProcess();          //  Registra il processo in osgm acquisendo l'identificatore
    //  Memorizza il destrittore della pipe creata per un suo utilizza futuro
    registeredProcess[idProcess].pipeToProcess=pipeFromKernel[1];

    prev=signal(SIGCHLD,stopHandle);      //  Installa un gestore del segnale SIGCHLD inviato dal
                                          // sistema se un processo figlio riceve un SIGSTOP

    printf("\n%-16.16s%d",filename,idProcess);

    testStop=1;
    //  Crea un processo in Unix che ospiterà il nuovo processo osgm
    if (id=fork()) { // Padre
        //  Memorizza il pid interno ad Unix per un uso futuro
        registeredProcess[idProcess].idProcess=id;
        //  Memorizza il nome del processo per un uso futuro
        strcpy(registeredProcess[idProcess].name,filename);
                }
                else
                { // Figlio
        char args[8][16];
        char *argv[10]={filename,args[0],args[1],args[2],args[3],args[4],args[5],args[6],args[7],NULL};
        //  Prepara gli argomenti da passare al processo creato
        sprintf(args[0],"%d",pipeToKernel);
        sprintf(args[1],"%d",pipeFromKernel[0]);
        sprintf(args[2],"%d",idProcess);
        sprintf(args[3],"%d",semaphoreKernel);
        sprintf(args[4],"%d",pipeToXSimulator);
        sprintf(args[5],"%d",handleCore);
        sprintf(args[6],"%p",baseMemory);
        sprintf(args[7],"%d",semaphoreMemory);
        //  Esegue il processo osgm sostituendo il processo creato dalla fork()
        execv(filename,argv);
        //  Se il processo arriva qui si è generato un errore
        return -1;
                }
    while(testStop);       //  Attende che il processo si blocchi
    signal(SIGCHLD,prev);  //  Ripristina il vecchio gestore del segnale SIGCHLD
    return 0;
}
#endif __SIMULATOR__


//  Questa funzione permette di stabilire se è possibile creare nuovi processi
int isProcessTableFull()
{
    //  Se il puntatore allo stack contenente gli slot liberi nella tabella dei processi
    // è arrivato alla base significa che la tabella è piena.
    if (processStackPointer < 0) return -1;
    return 0;
}

//  Questa funzione permette di registrare un processo in osgm
int registerProcess()
{
    int id;                                        //  Ospiterà il pid del processo registrato
```

```
    if (processStackPointer < 0) return -1;          //  Verifica che la tabella non sia piena
    id=processStack[processStackPointer--];          //  Acquisisce il numero di uno slot libero nella
                                                     // tabella dei processi.
    registeredProcess[id].status=READY;              //  Indica che inizialmente il processo è READY
                                                     // e pronto ad essere eseguito.
    registeredProcess[id].decodeWait=0;              //  Indica che non ci sono messaggi da decodificare
    registeredProcess[id].used=1;                    //  Segna l'elemento come utilizzato
    registeredProcess[id].priority=InitialPriority;  //  Setta la priorità iniziale del processo
    registeredProcess[id].next=-1;                   //  Inizializza i puntatori della lista doppiamente
    registeredProcess[id].prev=-1;                   // concatenata implementata attraverso l'array.
    insertProcess(id);                               //  Inserisce il processo creato nello scheduler
    return id;                                       //  Ritorna l'identificatore del processo
}

//  Questa funzione permette di eliminare un processo dal sistema
int unregisterProcess(id)
    int id;     //  Indica il pid del processo da eliminare
{
    //  Verifica che la tabella dei processi non sia vuota
    if (processStackPointer >= NumberOfRegisteredProcess) return -1;
    //  Verifica che il pid indichi un elemento non vuoto
    if (!registeredProcess[id].used) return -1;
    //  Inserisce il pid tra gli slot liberi dello stack processStack
    processStack[processStackPointer++]=id;
    //  Segna l'elemento come non usato
    registeredProcess[id].used=0;
    //  Cancella l'elemento dalla lista dello scheduler
    deleteProcess(id);
    return 0;
}

//  Questa funzione inizializza lo stack degli elementi liberi nella tabella dei processi
void initProcessStack()
{
    int i;

    //  Inizializza il puntatore dello stack all'ultimo elemento introdotto
    processStackPointer=NumberOfRegisteredProcess-1;
    //  Introduce tutti gli elementi della tabella dei processi nello stack
    // in ordine decrescente
    for (i=0;i<NumberOfRegisteredProcess;i++) processStack[i]=NumberOfRegisteredProcess-i-1;
}

// Inserisce il processo nella lista di schedulazione
void insertProcess(id)
    int id;  //  Indica il numero di processo da inserire nella lista dello scheduler
{
    //  Contiene la testa della lista
    int i=schedulerListHead;
    //  Controlla che la lista non sia vuota
    if (i < 0) { schedulerListHead=id; return; }
    //  Scandisce tutta la lista alla ricerca di un elemento che abbia una priorità più
    // bassa. La testa della lista punta all'elemento a priorità più alta
    while(registeredProcess[i].next >= 0 &&
          registeredProcess[registeredProcess[i].next].priority <= registeredProcess[id].priority)
          i=registeredProcess[i].next;
    //  Se elemento non è l'ultimo il puntatore dell'elemento che segue viene settato al nuovo
    // elemento
    if (registeredProcess[i].next >= 0) registeredProcess[registeredProcess[id].next].prev=id;
    //  Vengono settati i puntatori del nuovo elemento
    registeredProcess[id].next=registeredProcess[i].next;
    registeredProcess[id].prev=i;
    //  Viene modificato il puntatore dell'elemento trovato in modo che punti all'elemento introdotto
    registeredProcess[i].next=id;
}

// Elimina il processo dalla lista di schedulazione
void deleteProcess(id)
    int id;  //  Indica il numero di processo da eliminare dalla lista dello scheduler
{
    //  Se l'elemento non è l'ultimo viene modificato il puntatore dell'elemento successivo
    // facendolo puntare all'elemento precedente
    if (registeredProcess[id].next >= 0)
        registeredProcess[registeredProcess[id].next].prev=registeredProcess[id].prev;
    //  Se l'elemento non è il primo viene modificato il puntatore dell'elemento precedente
    // facendolo puntare all'elemento successivo. Se l'elemento è il primo la testa
    // punterà al prossimo elemento.
    if (registeredProcess[id].prev >= 0)
        registeredProcess[registeredProcess[id].prev].next=registeredProcess[id].next;
        else
        schedulerListHead=registeredProcess[id].next;
    registeredProcess[id].prev=-1;
    registeredProcess[id].next=-1;
}

//  Questa funzione si occupa di intercettare i segnali di trigger per riattivare il timer
// che chiamerà successivamente lo scheduler
void retrigger(signum)
    int signum;        //  Numero del segnale intercettato (SIGTRIG)
```

```
{
    alarm(QUANTUM);    //  Riavvia il segnale di allarme per attivare lo scheduler
}

//  Questa funzione si occupa di smistare le richieste inoltrate al kernel
void kernel(signum)
    int signum;        //  Numero del segnale intercettato (SIGSCHE, SIGMESS)
{
    pthread_t id;      //  Memorizza temporaneamente il tid del thread creato

    switch(signum) {
        case SIGSCHE:
            //  Crea un thread per la gestione dello scheduler
            pthread_create(&id,NULL,threadScheduler,NULL);
            break;
        case SIGMESS:
            //  Crea un thread per la gestione del messager
            pthread_create(&id,NULL,threadMessager,NULL);
            break;
               }
}

//  Funzione principale del thread addetto allo scheduling
void *threadScheduler(arg)
    void *arg;  //  Argomenti passati al thread
{
    //  Assicura la mutua esclusione nell'accesso alle strutture dati del kernel
    down(semaphoreKernel);
    //  Modifica lo stato del processo corrente
    if (registeredProcess[currentProcess].status != BLOCKED) setProcessStatus(currentProcess,READY);
    scheduler();  //  Avvia lo scheduler
    //  Assicura la mutua esclusione nell'accesso alle strutture dati del kernel
    up(semaphoreKernel);
}

// Questa funzione esplica tutte le funzionalità dello scheduler
void scheduler()
{
    int i;

    #ifdef __SIMULATOR__
    kill(registeredProcess[currentProcess].idProcess,SIGSTOP);   //  Blocca il processo correntemente in
    #endif __SIMULATOR__                                         // in esecuzione

    //  Verifica che ci sia almeno un processo READY, pronto ad essere eseguito
    i=schedulerListHead;
    while(registeredProcess[i].status == BLOCKED) {
        i=registeredProcess[i].next;
        if (i == -1) return;
       };

    //  Cancella il processo corrente dalla testa della lista e lo inserisce in una posizione
    // dipendente dalla priorità del processo e di tutti gli altri.
    do {
        deleteProcess(currentProcess);
        insertProcess(currentProcess);
        currentProcess=schedulerListHead;
        //  Se il processo in testa è READY viene eseguito
       } while(registeredProcess[currentProcess].status == BLOCKED);
    //  Esegue il processo trovato
    resumeProcess(currentProcess);
    //  Riavvia l'allarme inviando il segnale di trigger
    kill(kernelPid,SIGTRIG);
}

//  Questa funzione si occupa di riprendere l'esecuzione di processo precedentemente interrotto
void resumeProcess(id)
    int id;   //  Contiene l'identificatore del processo
{
    setProcessStatus(id,RUNNING);                  //  Cambia lo stato del processo in RUNNING, in esecuzione
    #ifdef __SIMULATOR__
    kill(registeredProcess[id].idProcess,SIGCONT);  //  Invia il segnale SIGCONT al processo per rimandarlo
    #endif __SIMULATOR__                            // in esecuzione.
}

//  Funzione principale del thread addetto al messaging
void *threadMessager(arg)
    void *arg;  //  Argomenti passati al thread
{
    //  Assicura la mutua esclusione nell'accesso alle strutture dati del kernel
    down(semaphoreKernel);
    messager();
    //  Assicura la mutua esclusione nell'accesso alle strutture dati del kernel
    up(semaphoreKernel);
}

// Questa funzione esplica tutte le funzionalità del messager
void messager()
{
```

```
char body[MaxMessageBodyLength];  //  Questo buffer contiene il corpo del messaggio inviato al kernel
int n,i;
int message;                      //  Questa variabile memorizza il numero del messaggio ricevuto
int nmessages;                    //  Questa variabile memorizza il numero di messaggi ricevuti
int idSource=-1;                  //  Questa variabile memorizza l'identificativo del mittente

#ifdef __SIMULATOR__
struct stat info;                 //  Questa strutture permette di leggere la quantità di
                                  // informazioni presenti in una qualunque pipe.
struct sembuf o;                  //  Questa struttura permette di leggere il valore del semaforo
                                  // usato come variabile condivisa tra tutti i processi del
                                  // simulatore. Tale variabile memorizza il numero di messaggi in
                                  // in attesa.
#endif __SIMULATOR__
SharedData *sharedData;           //  Questa struttura permette di scrivere e leggere blocchi di
                                  // informazioni nella memoria permanente.

#ifdef __SIMULATOR__
//  Acquisisce dalla variabile condivisa il numero di messaggi in attesa
nmessages=semctl(semaphoreKernel,1,GETVAL,NULL);
#endif __SIMULATOR__

for (i=0;i<nmessages;i++) {
//  Legge uno per volta i messaggi inviati dai processi
receiveMessage(ALLPROCESSES,&message,body,&n,&idSource);
//  Verifica che non sia stato richiesto un servizio
if (message & (NumberOfRegisteredMessage*2)) {
                        //  Blocca il processo che ha richiesto il servizio
                        setProcessStatus(idSource,BLOCKED);
                        //  Se il processo corrente è uguale al processo che ha
                        // richiesto il servizio viene bloccato ed uno nuovo
                        // viene scelto.
                        if (idSource == currentProcess) scheduler();
                        //  Viene filtrato il numero di messaggio normale
                        message &= ~(NumberOfRegisteredMessage*2);
                                        }
//  Riconosce i diversi messaggi facendo distinzione tra i messaggi diretti al kernel
// ed ai processi.
switch(message) {
        //  Questo messaggio viene spedito al kernel quando si vuole registrare un messaggio
        case KERNEL_REGISTER_MESSAGE:
            n=registerMessage(body);                    //  Individua un elemento libero
            registeredMessage[n].idProcess=idSource;    //  Memorizza il mittente del messaggio
            //  Se il processo è bloccato viene sbloccato
            if (registeredProcess[idSource].status == BLOCKED) setProcessStatus(idSource,READY);
            //  Viene spedita la risposta al processo
            sendMessage(idSource,KERNEL_ACK,&n,sizeof(int),IDKERNEL);
            break;
        //  Questo messaggio viene spedito al kernel quando si vuole decodificare un messaggio
        case KERNEL_DECODE_MESSAGE:
            //  Individua un elemento libero
            if ((n=decodeMessage(body)) < 0) {
                                        //  E' stato trovato un messaggio non ancora registrato
                                        undecodeMessages++;
                                        //  Pone in attesa di decodifica il processo
                                        registeredProcess[idSource].decodeWait=1;
                                        //  Registra il tipo di messaggio da decodificare
                                        strcpy(registeredProcess[idSource].undecodeMessage,body);
                                        //  Blocca il processo e se necessario viene scelto
                                        //  un nuovo processo
                                        setProcessStatus(idSource,BLOCKED);
                                        if (currentProcess == idSource) scheduler();
                                        break;
                                            }
            //  Se il processo è bloccato viene sbloccato
            if (registeredProcess[idSource].status == BLOCKED)  setProcessStatus(idSource,READY);
            //  Viene spedita la risposta al processo
            sendMessage(idSource,KERNEL_ACK,&n,sizeof(int),IDKERNEL);
            break;
        //  Questo messaggio viene spedito al kernel per avvertirlo che il processo attende
        // la ricezione di un messaggio.
        case KERNEL_RECEIVE_MESSAGE:
            //  Controlla se ci sono messaggi in attesa per il processo
            fstat(registeredProcess[idSource].pipeToProcess,&info);
            //  Ritorna se ci sono messaggi in attesa
            if (info.st_size) break;
            //  Blocca il processo
            #ifdef __SIMULATOR__
            kill(registeredProcess[idSource].idProcess,SIGSTOP);
            #endif
            //  Segna il processo come bloccato
            setProcessStatus(idSource,BLOCKED);
            //  Se è necessario sceglie un nuovo processo da eseguire
            if (currentProcess == idSource) scheduler();
            break;
        //  Questo messaggio viene spedito al kernel per avvertirlo che il processo vuole essere
        // bloccato
        case KERNEL_STOP_PROCESS_MESSAGE:
            //  Blocca il processo
```

```
                #ifdef __SIMULATOR__
                kill(registeredProcess[idSource].idProcess,SIGSTOP);
                #endif
                //  Segna il processo come bloccato
                setProcessStatus(idSource,BLOCKED);
                //  Se è necessario sceglie un nuovo processo da eseguire
                if (currentProcess == idSource) scheduler();
                break;
            //  Questo messaggio viene spedito al kernel quando si vuole scrivere nella memoria
            // permanente (FileSytem)
            case WRITE_SHARED_MEMORY:
                //  Permette di accedere alle informazioni presenti nel corpo del messaggio
                sharedData=(SharedData *) body;
                //  Scrive le informazioni puntate da sharedData->p nella memoria permanente
                n=writeBlock(sharedData->p,sharedData->n);
                //  Notifica alla parte grafica del simulatore che ci sono stati cambiamenti
                #ifdef __SIMULATOR__
                setSharedStatus();
                #endif __SIMULATOR__
                //  Se il processo è bloccato viene sbloccato
                if (registeredProcess[idSource].status == BLOCKED)  setProcessStatus(idSource,READY);
                //  Viene spedita la risposta al processo
                sendMessage(idSource,KERNEL_ACK,&n,sizeof(n),IDKERNEL);
                break;
            //  Questo messaggio viene spedito al kernel quando si vuole leggere dalla memoria
            // permanente (FileSytem)
            case READ_SHARED_MEMORY:
                //  Permette di accedere alle informazioni presenti nel corpo del messaggio
                sharedData=(SharedData *) body;
                //  Legge le informazioni dalla memoria permanente e li deposita nel buffer
                // puntato da sharedData->p
                n=readBlock(sharedData->p,sharedData->n);
                //  Notifica alla parte grafica del simulatore che ci sono stati cambiamenti
                #ifdef __SIMULATOR__
                setSharedStatus();
                #endif __SIMULATOR__
                //  Se il processo è bloccato viene sbloccato
                if (registeredProcess[idSource].status == BLOCKED) setProcessStatus(idSource,READY);
                //  Viene spedita la risposta al processo
                sendMessage(idSource,KERNEL_ACK,&n,sizeof(n),IDKERNEL);
                break;
            //  Tutti gli altri messaggi vengono rediretti ai processi
            default:
                //  Verifica che il numero del messaggio sia corretto
                if (message < 0 || message >= registeredMessagePointer) break;
                //  Se il processo è bloccato viene sbloccato
                if (registeredProcess[registeredMessage[message].idProcess].status == BLOCKED)
                        setProcessStatus(registeredMessage[message].idProcess,READY);
                //  Viene inoltrato il messaggio
                sendMessage(registeredMessage[message].idProcess,message,body,n,IDKERNEL);
                }
                        }
    //  Sottrae alla variabile condivisa che ospita il numero di messaggi in attesa il numero di
    // messaggi gestiti
    o.sem_op=-nmessages;
    o.sem_flg=0;
    o.sem_num=1;
    semop(semaphoreKernel,&o,1);
}

//  Questa funzione permette di registrare un messaggio nel sistema
int registerMessage(message)
    char *message;  // Contiene un puntatore ad una stringa Ascii-Z che identifica il messaggio
{
    int i;

    //  Verifica che non ci siano messaggi uguali già registrati nel sistema
    for (i=0;i<registeredMessagePointer;i++) if (!strcmp(registeredMessage[i].name,message)) return -1;

    //  Copia il messaggio nella tabella dei messaggi
    strcpy(registeredMessage[registeredMessagePointer].name,message);

    //  Verifica che non ci siano processi in attesa di questo messaggio
    if (undecodeMessages)
        //  Scandisce tutta la tabella dei processi alla ricerca dei processi che hanno
        // bisogno del messaggio
        for (i=0;i<NumberOfRegisteredProcess;i++)
            //  Controlla se l'elemento è usato
            if (registeredProcess[i].used &&
                //  Controlla se l'elemento ha messaggi da decodificare
                registeredProcess[i].decodeWait &&
                //  Controlla se il messaggio da decodificare è quello corrente
                !strcmp(registeredProcess[i].undecodeMessage,message)) {
                //  Elimina l'attesa del processo per messaggi da decodificare
                registeredProcess[i].decodeWait=0;
                //  Decrementa il numero di messaggi da decodificare
                undecodeMessages--;
                //  Spedisce la risposta al processo in attesa
                sendMessage(i,KERNEL_ACK,&registeredMessagePointer,sizeof(int),IDKERNEL);
```

```
                                                                }
    //  Ritorna il numero di messaggio registrato
    return registeredMessagePointer++;
}

//  Questa funzione permette di decodificare un messaggio, trasformando una stringa Ascii-Z in un
// identificativo.
int decodeMessage(message)
    char *message; //  Contiene un puntatore ad una stringa Ascii-Z che identifica il messaggio
{
    int i;

    //  Controlla se il messaggio è registrato
    for (i=0;i<registeredMessagePointer;i++) if (!strcmp(registeredMessage[i].name,message)) return i;
    return -1;
}

//  Questa funzione legge un blocco di informazioni dalla memoria permanente
int readBlock(buffer,n)
    void *buffer;         //  Contiene un puntatore al buffer di lettura
    int n;                //  Indica il numero di caratteri da leggere dalla memoria permanente
{
    int l;
    #ifdef __SIMULATOR__
    int handle;           //  Memorizza il descrittore di file con cui viene implementata la
    #endif __SIMULATOR__  // memoria permanente

    if (!n) return 0;     //  Ritorna se non ci sono caratteri da leggere
    //  Quando il puntatore di scrittura e di lettura coincidono non ci sono dati da leggere
    if (sharedMemoryRead == sharedMemoryWrite) return 0;
    //  Viene calcolato il numero massimo di caratteri che è possibile leggere
    if (sharedMemoryRead < sharedMemoryWrite) l=sharedMemoryWrite-sharedMemoryRead;
                                        else
                                        l=SHARED_MEMORY_LENGTH-sharedMemoryRead;
    if (n < l) l = n;     //  Viene aggiustata la quantità di informazione da leggere

    #ifdef __SIMULATOR__
    handle=open(SharedMemoryFile,O_RDONLY);  //  Viene aperto il file utilizzato per implementare
                                             // la memoria permanente
    if (handle == -1) return 0;              //  Verifica che il file è stato aperto
    lseek(handle,sharedMemoryRead,SEEK_SET); //  Viene posizionata la testina di lettura
    l=read(handle,buffer,l);                 //  Vengono letti i caratteri stabiliti
    close(handle);                           //  Il file viene chiuso
    #endif __SIMULATOR__
    sharedMemoryRead += l;                   //  Il puntatore di lettura viene aggiornato
    //  Viene verificato che il puntatore non superi il limite superiore imposto da SHARED_MEMORY_LENGTH
    if (sharedMemoryRead == SHARED_MEMORY_LENGTH) sharedMemoryRead=0;
    #ifdef __SIMULATOR__
    writePointers();                         //  Vengono scritti i puntatori sul disco
    sync();                                  //  Viene liberata la cache di scrittura
    #endif __SIMULATOR__
    return l+readBlock(buffer+l,n-l);        //  Richiama ricorsivamente la medesima funzione rilasciando
                                             // il numero di caratteri letti.
}

int writeBlock(buffer,n)
    void *buffer;         //  Contiene un puntatore al buffer di scrittura
    int n;                //  Indica il numero di caratteri da scrivere nella memoria permanente
{
    int l;
    #ifdef __SIMULATOR__
    int handle;           //  Memorizza il descrittore di file con cui viene implementata la
    #endif __SIMULATOR__  // memoria permanente

    if (!n) return 0;     //  Ritorna se non ci sono caratteri da scrivere
    //  Viene calcolato il numero massimo di caratteri che è possibile scrivere
    if (sharedMemoryRead <= sharedMemoryWrite) l=SHARED_MEMORY_LENGTH-sharedMemoryWrite;
                                         else
                                         l=sharedMemoryRead-sharedMemoryWrite;
    if (n < l) l = n;     //  Viene aggiustata la quantità di informazione da leggere
    #ifdef __SIMULATOR__
    //  Viene aperto il file utilizzato per implementare la memoria permanente
    handle=open(SharedMemoryFile,O_RDWR | O_CREAT,S_IREAD | S_IWRITE);
    lseek(handle,sharedMemoryWrite,SEEK_SET);   //  Viene posizionata la testina di lettura
    l=write(handle,buffer,l);                   //  Vengono scritti i caratteri stabiliti
    close(handle);                              //  Il file viene chiuso
    #endif __SIMULATOR__
    sharedMemoryWrite += l;                     //  Il puntatore di scrittura viene aggiornato
    //  Viene verificato che il puntatore non superi il limite superiore imposto da SHARED_MEMORY_LENGTH
    if (sharedMemoryWrite == SHARED_MEMORY_LENGTH) sharedMemoryWrite=0;
    //  Viene verificato che il puntatore di scrittura non superi il puntatore di lettura
    if (sharedMemoryWrite == sharedMemoryRead) sharedMemoryRead += SharedMemoryGranularity;
    //  Viene verificato che il puntatore non superi il limite superiore imposto da SHARED_MEMORY_LENGTH
    if (sharedMemoryRead >= SHARED_MEMORY_LENGTH) sharedMemoryRead -= SHARED_MEMORY_LENGTH;
    #ifdef __SIMULATOR__
    writePointers();                            //  Vengono scritti i puntatori sul disco
    sync();                                     //  Viene liberata la cache di scrittura
    #endif __SIMULATOR__
    return l+writeBlock(buffer+l,n-l);          //  Richiama ricorsivamente la medesima funzione
```

```
                                        // rilasciando il numero di caratteri letti.
}

//  Questa funzione cambia lo stato di un processo e notifica il cambiamento alla parte grafica
// del simulatore
void setProcessStatus(id,status)
    int id;      //  Indica il processo di cui si vuole cambiare lo stato
    int status;  //  Contiene il nuovo stato da affibiare al processo
{
    #ifdef __SIMULATOR__
    MessageXSimulator messageXSimulator;        //  Questa struttura permette di comunicare con la
                                                // parte grafica del simulatore.

    messageXSimulator.id=ID_PROCESS_REFRESH;
    *((int *) messageXSimulator.body)=status;   //  Memorizza il nuovo stato
    *(((int *) messageXSimulator.body)+1)=id;   //  Memorizza l'identificativo del mittente
    //  Memorizza il nome del processo
    strcpy(messageXSimulator.body+2*sizeof(int),registeredProcess[id].name);
    //  Spedisce il contenuto della struttura alla parte grafica
    if (pipeToXSimulator) write(pipeToXSimulator,&messageXSimulator,sizeof(MessageXSimulator));
    #endif __SIMULATOR__
    //  Modifica lo stato corrente del processo
    registeredProcess[id].status=status;
}

#ifdef __SIMULATOR__
//  Questa funzione permette di leggere dal file utilizzato per implementare la memoria
// permanente un blocco di caratteri per essere visualizzati dalla parte grafica
void readSharedBlock(seek,buffer)
    long seek;        //  Indica la posizione della testina di lettura
    char *buffer;     //  Contiene un puntatore al buffer di lettura
{
    int i,q,handle;
    struct stat info; //  Questa struttura viene utilizzata per acquisire informazioni
                      // sul file da leggere

    buffer[0];        //  Nel buffer di lettura viene inserita una stringa Ascii-Z vuota
    handle=open(SharedMemoryFile,O_RDONLY);     //  Viene aperto il file da leggere
    if (handle == -1) return;                   //  Verifica che il file è stato aperto
    fstat(handle,&info);                        //  Acquisisce informazioni sul file
    lseek(handle,seek,SEEK_SET);                //  Posiziona la testina di lettura
    q=read(handle,buffer,DEMOSTRATION_BLOCK);   //  Legge i caratteri
    //  Controlla se i caratteri letti sono insufficienti e decide se è il caso di
    // leggerli dall'inizio del file
    if (q < DEMOSTRATION_BLOCK && info.st_size == SHARED_MEMORY_LENGTH-1) {
                                                //  Posiziona la testina di lettura all'inizio del
                                                // file
                                                lseek(handle,0,SEEK_SET);
                                                //  Legge i caratteri necessari
                                                q += read(handle,buffer+q,DEMOSTRATION_BLOCK-q);
                                                                        }
    buffer[q]=0;      //  Termina la stringa Ascii-Z
    close(handle);    //  Chiude il file
    //  Elimina caratteri di controllo che in ogni caso non verrebbero stampati
    for (i=0;i<q;i++) if (buffer[i] < 0x20) buffer[i]=0x20;
}

//  Questa funzione notifica alla parte grafica del simulatore che ci sono stati cambiamenti
// nella gestione della meomoria permanente.
void setSharedStatus()
{
    MessageXSimulator messageXSimulator;        //  Questa struttura permette di comunicare con la
                                                // parte grafica del simulatore.

    messageXSimulator.id=ID_WRITE_SHARED_MEMORY_REFRESH;
    //  Formatta la stringa dimostrativa legata al puntatore di scrittura dalla
    // memoria permanente e la spedisce
    sprintf(messageXSimulator.body,"addr(write) 0x%6.6lx: ",sharedMemoryWrite);
    readSharedBlock(sharedMemoryWrite,messageXSimulator.body+strlen(messageXSimulator.body));
    if (pipeToXSimulator) write(pipeToXSimulator,&messageXSimulator,sizeof(MessageXSimulator));
    //  Formatta la stringa dimostrativa legata al puntatore di lettura dalla
    // memoria permanente e la spedisce
    messageXSimulator.id=ID_READ_SHARED_MEMORY_REFRESH;
    sprintf(messageXSimulator.body,"addr(read)  0x%6.6lx: ",sharedMemoryRead);
    readSharedBlock(sharedMemoryRead,messageXSimulator.body+strlen(messageXSimulator.body));
    if (pipeToXSimulator) write(pipeToXSimulator,&messageXSimulator,sizeof(MessageXSimulator));
}

// Questa funzione memorizza i puntatori alla memoria permanente in un file sul disco
void writePointers()
{
    int h;

    //  Apre il file
    h=open(SharedMemoryPointers,O_RDWR | O_CREAT,S_IREAD | S_IWRITE);
    //  Scrive il puntatore di lettura
    write(h,&sharedMemoryRead,sizeof(long));
    //  Scrive il puntatore di scrittura
    write(h,&sharedMemoryWrite,sizeof(long));
```

```
    //  Chiude il file
    close(h);
}
#endif __SIMULATOR__
```

Appendice A5. Codice sorgente: osgm.h

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.osgm.h
    Created by Antonino Sicali
*/

#ifndef _OSGM_H
#define _OSGM_H

#include "limits.h"

#define MessageLength                   16       //  Indica la dimensione in byte del nome
                                                 // associato ad un messaggio
#define NumberOfRegisteredMessage       32       //  Numero massimo di messaggi registarbili dal sistema
#define MaxMessageBodyLength            PIPE_BUF //  Indica la lunghezza massima del corpo di un messaggio
#define MemorySize                      256000   //  Dimensione della memoria comune ai processi

#ifndef NULL
#define NULL (void *) 0
#endif

#ifdef __SIMULATOR__
#define ID_MEMORY_REFRESH               0        //  Identica un'avvenuta modifica delle strutture dati del kernel
#define ID_PROCESS_REFRESH              1        //  Identica un'avvenuta modifica delle strutture dati del
                                                 // gestore della memoria comune
#define ID_READ_SHARED_MEMORY_REFRESH   2        //  Identica un'avvenuta modifica del puntatore di lettura dalla
                                                 // memoria permanente
#define ID_WRITE_SHARED_MEMORY_REFRESH 3         //  Identica un'avvenuta modifica del puntatore di scrittura
                                                 // nella memoria permanente

typedef struct {                                 //  Questa struttura viene utilizzata per spedire notifiche alla
                                                 // parte grafica del simulatore
    int id;                                      //  Contiene il tipo di notifica inoltrata
    char body[128];                              //  Contiene eventuali informazioni aggiuntive
} MessageXSimulator;

#endif __SIMULATOR__

typedef void (*messageHandle)(void *,int);       //  Tipo di dato utilizzato per definire gestori di messaggi

typedef struct {                                 //  Questa struttura viene utilizzata per definire array di
                                                 // gestori di messaggi
    messageHandle func;                          //  Contiene un puntatore ad un gestore di messaggi
    int id;                                      //  Contiene l'identificativo del messagio gestito dalla funzione
} MessageHandleStruct;                           // func()

#define MEMORY   "/mem/ram"                      //  Questo messaggio viene utilizzato per accedere alla memoria
                                                 // comune
#define SERIAL   "/io/serial"                    //  Questo messaggio viene utilizzato per accedere alla porta
                                                 // seriale
#define PARALLEL "/io/parallel"                  //  Questo messaggio viene utilizzato per accedere alla porta
                                                 // parallela
#define KEYBOARD "/io/keyboard"                  //  Questo messaggio viene utilizzato per accedere al tastierino
                                                 // numerico
#define LCD      "/io/lcd"                       //  Questo messaggio viene utilizzato per accedere a lcd

//  Messaggio MEMORY
#define OP_ALLOC   0                             //  Indica un'operazione di allocazione della memoria comune
#define OP_FREE    1                             //  Indica una deallocazione della memoria comune

typedef struct {                                 //  Questa struttura permette di richiedere i servizi al gestore
                                                 // della memoria comune
    int idMessage;                               //  Specifica l'identicativo del mittente
    int size;                                    //  Specifica la dimensione del blocco da allocare
    int op;                                      //  Specifica l'operazione da compiere
    char *p;                                     //  Specifica l'indirizzo del blocco di memoria da liberare
} MemoryQuery;

//  Messaggio SERIAL, PARALLEL
#define OP_WRITE   0                             //  Indica un'operazione di scrittura in una periferica di IO
#define OP_READ    1                             //  Indica un'operazione di lettura da una periferica di IO

typedef struct {                                 //  Questa struttura permette di richiedere i servizi al gestore
                                                 // delle periferiche di IO
    int idMessage;                               //  Specifica l'identicativo del mittente
    int size;                                    //  Specifica la dimensione del blocco di dati da leggere/scrivere
    int op;                                      //  Specifica l'operazione da compiere
    char *p;                                     //  Specifica l'indirizzo del buffer di lettura/scrittura
} IoQuery;

extern int idApplication;                        //  Nome di identificazione del processo nel sistema

//  Contiene il numero di messaggio con cui l'aaplicazione può ricevere messaggi
// da altre applicazioni. Tale valore viene utilizzato solamente se applicazione
// utilizza servizi in cui è necessario un messaggio di ritorno. Per esempio
// allocare memoria richiede che il gestore della memoria sappia a chi spedire
// il puntatore al blocco allocato.
extern char APPLICATION[];
```

```
//  Contiene il descrittore della pipe utilizzata per inviare una notifica di modifica
// di parametri alla parte grafica del simulatore
extern int pipeToXSimulator;

#ifndef __KERNEL__
//  Questa funzione permette di spedire un messaggio da e per il kernel.
void sendMessage(int message,void *body,int n);

//  Questa funzione permette di ricever un messaggio da e per il kernel.
void receiveMessage(int *message,void *body,int *n);

// Questa funzione permette di richiedere un servizio al sistema. Un servizio è costituito da
// dalla spedizione di un messaggio e dalla ricezione di una risposta, il tutto viene effettuato
// in un unica operazione atomica.
void queryService(int message,void *source,int n,void *drain);
#endif  __KERNEL__

//  Questa funzione permette di registrare un messaggio nel sistema
int registerMessage(char *message);

//  Questa funzione permette di decodificare un messaggio, trasformando una stringa Ascii-Z in un
// identificativo.
int decodeMessage(char *message);

// Questa funzione permette di bloccare il processo
void stopProcess(void);

// Questa funzione permette di leggere un blocco di dati dall'area permanente (FileSystem)
int readBlock(void *buffer,int n);

// Questa funzione permette di scrivere un blocco di dati nell'area permanente (FileSystem)
int writeBlock(void *buffer,int n);

//  Prima funzione chiamata all'avvio del processo. Ha le stessa semantica di main per i
// processi C/C++.
int osgmMain(void);

#endif  _OSGM_H
```

Appendice A6. Codice sorgente: semaphore.c

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.semaphore.c
    Created by Antonino Sicali
*/

#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/ipc.h>

//  Questa funzione permette di effettuare un'operazione di down su un semaforo
void down(sem)
    int sem;                    //  Contiene l'identificatore del semaforo su cui effettuare down
{
    struct sembuf o={0,-1,0};   //  Questa struttura contiene le informazioni necessari

    semop(sem,&o,1);            // alla funzione semop per effetture una down
}

//  Questa funzione permette di effettuare un'operazione di up su un semaforo
void up(sem)
    int sem;                    //  Contiene l'identificatore del semaforo su cui effettuare up
{
    struct sembuf o={0,1,0};    //  Questa struttura contiene le informazioni necessari

    semop(sem,&o,1);            // alla funzione semop per effetture una down
}
```

Appendice A7. Codice sorgente: linux.c

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.linux.c
    Created by Antonino Sicali
*/

#include "kernel.h"
#include "string.h"

#ifdef __SIMULATOR__
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#endif __SIMULATOR__

int idProcess=-1;
//  Contiene l'indirizzo base della memoria comune a tutti i processi
void *baseMemory=NULL;
//  Contiene l'identificativo del semaforo che assicura la mutua esclusione sulla
// memoria condivisa.
int semaphoreMemory;
//  Contiene il descrittore associato con il segmento condiviso tra tutti i processi del
// simulatore.
int handleCore;
//  Contiene il descrittore della pipe utilizzata per inviare messaggi al kernel
int pipeToKernel;
//  Contiene il descrittore della pipe utilizzata per ricevere messaggi dal kernel
int pipeFromKernel;

#ifndef __KERNEL__
// Contiene il process id del kernel per permettere la comunicazione con il kernel
int idParent;
//  Contiene il numero di messaggio con cui l'aaplicazione può ricevere messaggi
// da altre applicazioni. Tale valore viene utilizzato solamente se applicazione
// utilizza servizi in cui è necessario un messaggio di ritorno. Per esempio
// allocare memoria richiede che il gestore della memoria sappia a chi spedire
// il puntatore al blocco allocato.
int idApplication;
//  Contiene il descrittore della pipe utilizzata per inviare una notifica di modifica
// di parametri alla parte grafica del simulatore
int pipeToXSimulator;
//  Contiene l'identificativo del semaforo utilizzato come variabile condivisa che memorizza
// il numero di messaggi inviati attraverso la pipe 'pipeToKernel'. L'identificativo individua
// un array di due semafori di cui solo il secondo viene utilizzato come variabile condivisa.
int semaphoreKernel;

//  Ereditata da tutti i processi permette al simulatore di presettare parametri indispensabili per i processi:
// argv[1] contiene il numero di descrittore associato alla pipe di scrittura verso il processo padre (kernel).
// argv[2] contiene il numero di descrittore associato alla pipe di lettura dal processo padre (kernel).
// argv[3] contiene il pid del processo
// argv[4] contiene l'identificativo del semaforo utilizzato come variabile condivisa per memorizzare il numero
//         di messaggi inviati al kernel.
// argv[5] contiene il descrittore della pipe di notifica alla parte grafica del simulatore
// argv[6] contiene il descrittore associato al segmento codiviso tra tutti i processi del simulatore
// argv[7] contiene un puntatore alla base della memoria codivisa tra tutti i processi del simulatore
// argv[8] contiene l'identificativo del semaforo che assicura la mutua esclusione nell'accesso alla memoria
//         condivisa
int main(argc,argv)
    int   argc;   //  Contiene il numero di parametri ricevuti dal processo padre
    char *argv[]; //  Contiene un puntatore ad un array di stringhe Ascii-Z che rappresentano
                  // i parametri di ingresso.
{
    idParent=getppid();
    //  Verifica che il numero di parametri sia corretto
    if (argc != 9) return -1;
    //  Converte i parametri testuali in formato binario
    if (!sscanf(argv[1],"%d",&pipeToKernel)) return -1;
    if (!sscanf(argv[2],"%d",&pipeFromKernel)) return -1;
    if (!sscanf(argv[3],"%d",&idProcess)) return -1;
    if (!sscanf(argv[4],"%d",&semaphoreKernel)) return -1;
    if (!sscanf(argv[5],"%d",&pipeToXSimulator)) return -1;
    if (!sscanf(argv[6],"%d",&handleCore)) return -1;
    if (!sscanf(argv[7],"%p",&baseMemory)) return -1;
    if (!sscanf(argv[8],"%d",&semaphoreMemory)) return -1;
    //  Lega insieme l'indirizzo base della memoria condivisa e il descrittore associato al segmento
    // condiviso.
    shmat(handleCore,baseMemory,0);
    //  Blocca il processo lasciando il controllo allo scheduler
    raise(SIGSTOP);
    //  Passa il controllo al processo osgm
    return osgmMain();
}
#endif __KERNEL__
```

```
//  Questa funzione permette di spedire un messaggio da e per il kernel.
#ifdef __KERNEL__
void sendMessage(id,message,body,n,idSource)
    int   id;       //  Contiene l'identificativo del processo a cui è indirizzato il messaggio
    int   message;  //  Contiene il numero di messaggio inviato
    void *body;     //  Contiene un puntatore al corpo del messaggio
    int   n;        //  Contiene la lunghezza del corpo del messaggio
    int   idSource; //  Contiene l'identificativo del processo da cui è stato inviato il messaggio
#else
void sendMessage(message,body,n)
    int   message;  //  Contiene il numero di messaggio inviato
    void *body;     //  Contiene un puntatore al corpo del messaggio
    int   n;        //  Contiene la lunghezza del corpo del messaggio
#endif __KERNEL__
{
    #ifdef __KERNEL__
    int i=0,m=NumberOfRegisteredMessage;

    if (id != ALLPROCESSES) m=(i=id)+1;
    //  Spedisce a tutti i processi nell'intervallo [i..m-1] il messaggio
    for (i=i;i<m;i++) {
        //  Verifica che il processo di identificativo 'i' sia registrato nel sistema
        if (!registeredProcess[i].used) continue;
        //  Acquisisce il descrittore della pipe di scrittura verso il processo
        pipeToKernel=registeredProcess[i].pipeToProcess;
    #else
    int idSource=idProcess;   //  Acquisisce l'identificativo del mittente
    struct sembuf o={1,1,0};  //  Inizializza la stuttura che permette di incrementare il semaforo
    #endif __KERNEL__         // utilizzato come variabile condivisa.
        write(pipeToKernel,&message,sizeof(int));  //  Scrive il numero di messaggio nella pipe
        write(pipeToKernel,&n,sizeof(int));        //  Scrive il numero di byte di cui il corpo è composto
        write(pipeToKernel,body,n);                //  Scrive il corpo del messaggio nella pipe
        write(pipeToKernel,&idSource,sizeof(int)); //  Scrive l'identificativo del mittente
                #ifdef __KERNEL__
                }           // Necessario per chiudere l'iterazione
                #else
                semop(semaphoreKernel,&o,1);     //  Incrementa il semaforo
                kill(idParent,SIGMESS);          //  Avverte il messager del kernel che un messaggio
                #endif __KERNEL__                // è stato spedito.
}

//  Questa funzione permette di ricever un messaggio da e per il kernel.
#ifdef __KERNEL__
void receiveMessage(id,message,body,n,idSource)
    int   id;       //  Contiene l'identificativo del processo da cui si aspetta un messaggio
    int  *message;  //  Contiene il numero di messaggio ricevuto
    void *body;     //  Contiene un puntatore al corpo del messaggio
    int  *n;        //  Contiene la lunghezza del corpo del messaggio
    int  *idSource; //  Contiene l'identificativo del processo da cui è stato ricevuto il messaggio
#else
void receiveMessage(message,body,n)
    int  *message;  //  Contiene il numero di messaggio ricevuto
    void *body;     //  Contiene un puntatore al corpo del messaggio
    int  *n;        //  Contiene la lunghezza del corpo del messaggio
#endif __KERNEL__
{
    #ifdef __KERNEL__
    // Verifica se il messaggio si attende da un processo in particolare
    if (id == ALLPROCESSES) pipeFromKernel=pipeFromProcesses; else return;
    #else
    int i;             //  Variabile di supporto per memorizzare l'identificativo del mittente
    int *idSource=&i;  //  Puntatore ad una locazione che memorizzerà l'identificativo del mittente

    sendMessage(KERNEL_RECEIVE_MESSAGE,NULL,0);  //  Avverte il kernel che si sta eseguendo un'istruzione
    #endif __KERNEL__                            // bloccante.

    read(pipeFromKernel,message,sizeof(int));    //  Legge il numero del messaggio
    read(pipeFromKernel,n,sizeof(int));          //  Legge la lunghezza del corpo del messaggio
    read(pipeFromKernel,body,*n);                //  Legge il corpo del messaggio
    read(pipeFromKernel,idSource,sizeof(int));   //  Legge l'identificativo del mittente
}

//  Le funzioni di seguito riportate costituiscono l'interfaccia che possono utilizzare i processi
// per comunicare con il kernel, e con gli altri processi.

#ifndef __KERNEL__
//  Questa funzione permette di registrare un messaggio nel sistema
int registerMessage(message)
    char *message;  // Contiene un puntatore ad una stringa Ascii-Z che identifica il messaggio
{
    int m,n,id;
    //  Richiede il servizio di registrazione di un messaggio al kernel
    queryService(KERNEL_REGISTER_MESSAGE,message,strlen(message)+1,&id);
    //  L'intero ritornato nella variabile 'id' identifica il messaggio nel sistema e
    // potrà essere usato per spedire messaggi di classe 'message'.
    return id;
}
//  Questa funzione permette di decodificare un messaggio, trasformando una stringa Ascii-Z in un
```

```
// identificativo.
int decodeMessage(message)
    char *message;
//  Contiene un puntatore ad una stringa Ascii-Z che identifica il messaggio
{
    int m,n,id,s;
    //  Richiede il servizio di decodifica di un messaggio al kernel
    queryService(KERNEL_DECODE_MESSAGE,message,strlen(message)+1,&id);
    //  L'intero ritornato nella variabile 'id' identifica il messaggio nel sistema e
    // potrà essere usato per spedire messaggi di classe 'message'.
    return id;
}

// Questa funzione permette di bloccare il processo
void stopProcess()
{
    sendMessage(KERNEL_STOP_PROCESS_MESSAGE,NULL,0);
}

// Questa funzione permette di richiedere memoria comune a tutti i processi
void *allocMemory(size)
    int size; //  Indica la dimensione del blocco da allocare
{
    //  Questa struttura permette di inviare i dati necessari al gestore della memoria.
    MemoryQuery mq={idApplication,size,OP_ALLOC,NULL};
    static int message=-1;  //  Questa variabile memorizza l'identificativo del messaggio che permette di
    int n,m;                // accedere alla memoria comune.
    char *newMemoryBlock;   //  Memorizza temporaneamente l'indirizzo del nuovo blocco di memoria.

    // Inizializza l'identificativo del messaggio che permette di accedere alla memoria comune
    if (message < 0) message=decodeMessage(MEMORY);
    // Spedisce il messaggio al gestore della memoria e si blocca in attesa della risposta
    queryService(message,&mq,sizeof(MemoryQuery),&newMemoryBlock);
    return newMemoryBlock;
}

// Questa funzione permette di rilasciare memoria comune a tutti i processi precedentemente allocata
void freeMemory(mem)
    void *mem;  //  Contiene un puntatore al blocco da deallocare
{
    MemoryQuery mq={0,0,OP_FREE,mem};
    static int message=-1;  //  Questa variabile memorizza l'identificativo del messaggio che permette di
    int n,m;                // accedere alla memoria comune.

    // Inizializza l'identificativo del messaggio che permette di accedere alla memoria comune
    if (message < 0) message=decodeMessage(MEMORY);
    // Spedisce il messaggio al gestore della memoria e si blocca in attesa della risposta
    sendMessage(message,&mq,sizeof(MemoryQuery));
}

// Questa funzione permette di leggere un blocco di dati dall'area permanente (FileSystem)
int readBlock(buffer,n)
    void *buffer;           //  Contiene un puntatore al buffer contenente i dati da leggere
    int n;                  //  Indica la dimensione del blocco di dati da leggere
{
    int m,q,s;
    SharedData a={buffer,n}; //  Questa struttura permette di inviare i dati necessari
                             // al kernel per leggere il blocco di dati.

    // Spedisce il messaggio al kernel e si blocca in attesa della risposta
    queryService(READ_SHARED_MEMORY,&a,sizeof(SharedData),&q);
    return q;
}

// Questa funzione permette di scrivere un blocco di dati nell'area permanente (FileSystem)
int writeBlock(buffer,n)
    void *buffer;           //  Contiene un puntatore al buffer contenente i dati da scrivere
    int n;                  //  Indica la dimensione del blocco di dati da scrivere
{
    int m,q,s;
    SharedData a={buffer,n}; //  Questa struttura permette di inviare i dati necessari
                             // al kernel per scrivere il blocco di dati.

    // Spedisce il messaggio al kernel e si blocca in attesa della risposta
    queryService(WRITE_SHARED_MEMORY,&a,sizeof(SharedData),&q);
    return q;
}

// Questa funzione permette di richiedere un servizio al sistema. Un servizio è costituito da
// dalla spedizione di un messaggio e dalla ricezione di una risposta, il tutto viene effettuato
// in un unica operazione atomica.
void queryService(message,source,n,drain)
    int   message;  //  Contiene il numero di messaggio inviato
    void *source;   //  Contiene un puntatore al corpo del messaggio
    int   n;        //  Contiene la lunghezza del corpo del messaggio
    void *drain;    //  Contiene un puntatore al buffer che ospiterà la risposta
{
    int idSource;                   //  Conterrà l'identificativo del mittente una volta ricevuta
                                    // la risposta.
```

```
    char body[MaxMessageBodyLength];  //  Questo buffer viene utilizzato per ospitare la risposta se
                                      // il puntatore al buffer di destinazione 'drain' è nullo

    // Spedisce il messaggio al sistema e si blocca in attesa della risposta
    sendMessage(message | (NumberOfRegisteredMessage*2),source,n);
    // Legge il numero di messaggio di ritorno
    read(pipeFromKernel,&message,sizeof(int));
    // Legge la dimensione della risposta
    read(pipeFromKernel,&n,sizeof(int));
    // Legge il corpo della risposta riponendola nel buffer di destinazione o cestinandola
    if (drain) read(pipeFromKernel,drain,n); else read(pipeFromKernel,body,n);
    // Legge l'identificativo del mittente
    read(pipeFromKernel,&idSource,sizeof(int));
}
#endif __KERNEL__
```

Appendice A8. Codice sorgente: memory.h

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.memory.h
    Created by Antonino Sicali
*/

#ifndef _MEMORY_H
#define _MEMORY_H

#include "osgm.h"

#define MINIMAL_BLOCK_SIZE  4096  //  Indica la dimensione minima dei blocchi di memoria

#define NMESSAGE            1     //  Indica il numero di messaggi che l'applicazione registrerà

typedef struct {                  //  Questa struttura dati viene utilizzata per operare sui blocchi di memoria
    char *p;                      //  Questo puntatore punta al prossimo blocco di memoria libera. Può
                                  // assumere il valore NULL se il blocco è l'ultimo, oppure -1 se il blocco
                                  // è costituito da memoria utilizzata.
    long size;                    //  Questa variabile indica la dimensione del blocco
} Node;

extern int handleCore;
extern void *baseMemory;

//  Questa funzione si occupa di gestire le richieste di memoria dinamica da parte dei processi.
//  La memoria allocata è comune a tutti i processi e può essere utilizzata come buffer di
// interscambio per grosse quantità di dati, in alternativa si dovrebbe utilizzare il corpo
// di un messaggio.
void memRam(void *body,int n);

//  Questa funzione permette di allocare un nuovo blocco di memoria. L'algoritmo utilizzato
// permette di trovare il primo blocco (FirstFit), i blocchi liberi vengono rintracciati per
// mezzo di una lista, i cui elementi vengono memorizzati all'inizio di ciscun blocco.
//  Un blocco allocato possiede all'inizio un struttura di tipo Node che memorizza, a differenza
// di un blocco libero, solamente la dimensione.
void *findBlock(int size);

// Questa funzione permette di liberare un blocco di memoria precedentemente allocato
void deleteBlock(char *mem);

//  Questa funzione inizializza l'area di memoria comune ai processi, creando un unico blocco libero.
inline void initMemory(mem)
    void *mem;  //  Contiene un puntatore alla base della memoria comune
{
    ((Node *) mem)->p=0;              //  Una struttura di tipo Node contiene un puntatore al prossimo
    ((Node *) mem)->size=MemorySize;  // blocco e un campo size che indica la lunghezza del blocco
                                      // corrente a partire dall'indirizzo della struttura di tipo Node.
}

#endif  _MEMORY_H
```

Appendice A9. Codice sorgente: memory.c

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.memory.c
    Created by Antonino Sicali
*/

#include "osgm.h"
#include "memory.h"

void *firstBlock;

//  Questa variabile contiene l'identificativo del semaforo utilizzato per assicurare
// l'accesso esclusivo alla memoria comune ai processi. In particolare il problema
// dell'accesso condiviso si genera tra l'applicazione grafica di monitoraggio 'xosgm'
// e il gestore della memoria. Può succedere che il gestore vada a leggere blocchi di
// memoria inesistenti a causa delle azioni di scrittura in più fasi del gestore,
// causando un qualche tipo di 'Segment Fault'.
extern int semaphoreMemory;

//  Nome di identificazione del processo nel sistema
char APPLICATION[]="Memory";

//  Prima funzione chiamata all'avvio del processo. Ha le stessa semantica di main per i
// processi C/C++.
int osgmMain()
{
    char body[MaxMessageBodyLength];          //  Viene utilizzato per memorizzare temporaneamente il corpo
    int n,i;                                  // del messaggio in arrivo. L'identificativo numerico del
    int message;                              // del messaggio verrà memorizzato nella variabile 'message'
    MessageHandleStruct messages[NMESSAGE];   //  Questa struttura memorizza i gestori dei messaggio registrati
                                              // nelle strutture dati del Kernel per mezzo della funzione
                                              // 'registerMessage'.

    initMemory(baseMemory);                   //  Inizializza l'area di memoria comune ai processi, creando
    firstBlock=baseMemory;                    // un unico blocco libero.

    messages[0].id=registerMessage(MEMORY);   //  Registra nel sistema un messaggio di gestione della memoria
    messages[0].func=memRam;                  // comune ai processi.

    while(1) {
    receiveMessage(&message,&body,&n);        //  Attende, ponendosi in stato di 'Blocked', un nuovo messaggio.
    for (i=0;i<NMESSAGE;i++)                  //  Ricerca il gestore del messaggio arrivato.
        if (message == messages[i].id) {
            messages[i].func(body,n);         //  Chiama il gestore del messaggio inoltrandone il corpo.
            break;
                                      }
         }
}

//  Questa funzione si occupa di gestire le richieste di memoria dinamica da parte dei processi.
//  La memoria allocata è comune a tutti i processi e può essere utilizzata come buffer di
// interscambio per grosse quantità di dati, in alternativa si dovrebbe utilizzare il corpo
// di un messaggio.
void memRam(body,n)
    void *body;          //  Punta ad una area di memoria contenente il corpo del messaggio
    int   n;             //  Contiene il numero di caratteri significativi contenuti nel buffer
{
    MemoryQuery *m=body; //  Prepara un puntatore del tipo opportuno per accedere in modo semplice
    char *newMemoryBlock;//  Memorizza temporaneamente l'indirizzo del nuovo blocco di memoria.

    #ifdef __SIMULATOR__
    MessageXSimulator messageXSimulator; //  Questa struttura dati permette di comunicare con la
                                         // parte grafica del simulatore per avvertirlo che le
                                         // informazioni visualizzate devono essere aggiornate.

    down(semaphoreMemory);               //  Assicura la mutua esclusione sulla memoria comune a
                                         // tutti i processi.
    #endif __SIMULATOR__

    switch(m->op) {
        case OP_ALLOC:
            newMemoryBlock=findBlock(m->size);
            sendMessage(m->idMessage,&newMemoryBlock,sizeof(char *));
            break;
        case OP_FREE:
            deleteBlock(m->p);
            break;
                }

    #ifdef __SIMULATOR__
    up(semaphoreMemory);                 //  Assicura la mutua esclusione sulla memoria comune a
                                         // tutti i processi.
    messageXSimulator.id=ID_MEMORY_REFRESH;  // Avverte la parte grafica del simulatore che la memoria
                                         // comune ha subito una modifica.
    if (pipeToXSimulator) write(pipeToXSimulator,&messageXSimulator,sizeof(MessageXSimulator));
    #endif __SIMULATOR__
}
```

```
//  Questa funzione permette di allocare un nuovo blocco di memoria. L'algoritmo utilizzato
// permette di trovare il primo blocco (FirstFit), i blocchi liberi vengono rintracciati per
// mezzo di una lista, i cui elementi vengono memorizzati all'inizio di ciscun blocco.
//  Un blocco allocato possiede all'inizio un struttura di tipo Node che memorizza, a differenza
// di un blocco libero, solamente la dimensione.
void *findBlock(size)
    int size; // Indica la dimenzione del blocco di memoria richiesto
{
    Node *this,*prev=NULL,*next;       //  Alla fine della ricerca si avranno tre informazioni:
                                       // indirizzo del blocco precedente, corrente e successivo.
                                       //  In base a tali informazioni si agirà di conseguenza.
    size += sizeof(Node);              //  Ogni blocco possiede all'inizio una struttura di tipo Node
                                       // quindi è necessario aumentare la richiesta della dimensione
                                       // di tale struttura.
    //  Corregge la dimensione del blocco se il blocco allocato è troppo piccolo
    if (size < MINIMAL_BLOCK_SIZE) size=MINIMAL_BLOCK_SIZE;
    this=firstBlock;                   //  La variabile firstBlock memorizza la testa della lista
                                       // e punta al primo blocco libero. Se firstBlock è un puntatore
                                       // nullo allora non ci sono blocchi liberi.

    if (!this) return NULL;            //  Ritorna poichè non cè memoria libera

    while(this->size < size) {         //  Ricerca un blocco libero che sia abbastanza grande
        prev=this;                     //  Scandisce tutta la lista finchè non si arriva all'ultimo
        this = (Node *) this->p;       // blocco ritornando NULL.
        if (!this->p) return NULL;
       };

    //  Esegue un controllo sulla dimensione per decidere se è conveniente spezzare il blocco
    if (size+sizeof(Node)+MINIMAL_BLOCK_SIZE < this->size) { // Il blocco viene spezzato
                                       // Calcola la fine del nuovo blocco creato
                                       next=(Node *) (((char*) this)+size);
                                       // Crea un nuovo blocco alla fine del nuovo blocco
                                       next->p=this->p;
                                       next->size=this->size-size;
                                       // Inizializza la dimensione del blocco creato
                                       this->size=size;
                                                 }
                                                 else
                                                 {
                                       //  Il prossimo blocco diventa il successivo nella lista
                                       // del blocco trovato.
                                       next=(Node *) this->p;
                                                 }
    //  Controlla se il blocco precedente esiste
    if (!prev) firstBlock=next;        //  Fa puntare la testa della lista al blocco successivo
           else
           prev->p = (char *) next;    //  Lega il blocco precedente a quello successivo

    this->p=(char *) -1;               //  Segna il blocco come utilizzato
    // Ritorna un puntatore all'inizio effettivo del nuovo blocco
    return ((char *) this)+sizeof(Node);
}

// Questa funzione permette di liberare un blocco di memoria precedentemente allocato
void deleteBlock(mem)
    char *mem;  //  Contiene un puntatore all'area di memoria da liberare
{
    Node *prev=NULL,*this,*next;       //  Alla fine della ricerca si avranno tre informazioni:
                                       // indirizzo del blocco precedente, corrente e successivo.
                                       //  In base a tali informazioni si agirà di conseguenza.

    mem -= sizeof(Node);               //  Il blocco di memoria inizia prima di mem poichè include una
                                       // struttura di tipo Node nella parte iniziale;
    this=(Node *) mem;                 //  Viene inizializzato il blocco da inserire nella lista la cui
    next=firstBlock;                   // testa è contenuta in firstBlock.

    if (!next) {                       //  Controlla se la lista è vuota (assenza di memoria)
        firstBlock=mem;                //  La testa della lista punterà al blocco liberato, la dimensione
        this->p=NULL;                  // del blocco non verrà modificata, mentre il puntatore al prossimo
        return;                        // blocco verrà inizializzato a NULL per indicare che il blocco è
           }                           // è l'ultimo della lista.

    while(next->p < (char *) this) {   //  Se la lista non è vuota viene ricercata la posizione in cui
        prev=next;                     // inserire il blocco. I blocchi della lista sono ordinati per
        next = (Node *) prev->p;       // indirizzo.
       };

    this->p=(char *) next;             // Il blocco da liberare punterà al blocco successivo
    //  Se esiste un blocco precedente questo punterà al blocco da liberare altrimenti
    // sarà la testa della lista a puntarlo.
    if (prev) prev->p=(char *) this; else firstBlock=(char *) this;
    // Verifica se è possibile una fusione tra il blocco da liberare e quello successivo
    if (this->p+this->size == (char *) next) { this->size += next->size; this->p=next->p; }
    // Verifica se è possibile una fusione tra il blocco da liberare e quello precedente
    if (prev && prev->p+prev->size == (char *) this) { prev->size += this->size; prev->p=this->p; }
}
```

```
      /*
    Istituto Nazionale di Geofisica e Vulcanologia - Oseervatorio Etneo
    OSGM.io.h
    Created by Antonino Sicali
*/

#ifndef _IO_H
#define _IO_H

#include "osgm.h"

#define NMESSAGE 1      //  Indica il numero di messaggi che l'applicazione registrerà

//  Questa funzione si occupa di gestire l'IO verso (e dalla) porta seriale.
//  Il gestore in configurazione definitiva preleverà da un buffer interno i
// dati organizzati al carattere e li assembla in blocchi di lunghezza
// prefissata inoltrandoli successivamente al richiedente.
void serialIO(void *body,int n);

#endif  _IO_H

      /*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.io.c
    Created by Antonino Sicali
*/

#include "osgm.h"
#include "io.h"

//  Nome di identificazione del processo nel sistema
char APPLICATION[]="IO";

//  Prima funzione chiamata all'avvio del processo. Ha le stessa semantica di main per i
// processi C/C++.
int osgmMain()
{
    char body[MaxMessageBodyLength];         //  Viene utilizzato per memorizzare temporaneamente il corpo
    int n,i;                                 // del messaggio in arrivo. L'identificativo numerico del
    int message;                             // del messaggio verrà memorizzato nella variabile 'message'
    MessageHandleStruct messages[NMESSAGE];  //  Questa struttura memorizza i gestori dei messaggio registrati
                                             // nelle strutture dati del Kernel per mezzo della funzione
    messages[0].id=registerMessage(SERIAL);  // 'registerMessage'.
    messages[0].func=serialIO;

    while(1) {
    receiveMessage(&message,&body,&n);       //  Attende, ponendosi in stato di 'Blocked', un nuovo messaggio.
    for (i=0;i<NMESSAGE;i++)                 //  Ricerca il gestore del messaggio arrivato.
        if (message == messages[i].id) {
            messages[i].func(body,n);        //  Chiama il gestore del messaggio inoltrandone il corpo.
            break;
                                    }
         }
    return 0;
}

// Questa funzione si occupa di gestire l'IO verso (e dalla) porta seriale.
// Il gestore in configurazione definitiva preleverà da un buffer interno i
//  dati organizzati al carattere e li assembla in blocchi di lunghezza 'm->n'
//  inoltrandoli successivamente al richiedente.
void serialIO(body,n)
    void *body;          //  Punta ad una area di memoria contenente il corpo del messaggio
    int   n;             //  Contiene il numero di caratteri significativi contenuti nel buffer
{
    IoQuery *m=body;     //  Prepara un puntatore del tipo opportuno per accedere in modo semplice
    #ifdef __SIMULATOR__ // ed efficiente al corpo del messaggio.
    static int i=0;      //  Questa variabile è necessaria solo alla parte dimostrativa e quanto
    #endif __SIMULATOR__ // prima dovrà essere eliminata.

    switch(m->op) {
        case OP_READ:
            //  Questa porzione di codice è necessaria solo alla parte dimostrativa
            // e quanto prima dovrà essere eliminata.
            #ifdef __SIMULATOR__
            if (i) strcpy(m->p,"01-01-98 23:12:00 45120.67 a\xa\xd"); //  Permette al gestore della seriale
                   else                                               // di ritornare almeno due misure
                   strcpy(m->p,"03-02-99 23:12:10 45100.84 a\xa\xd"); // magnetiche differenti.
            i = 1-i;
            #endif __SIMULATOR__
            sendMessage(m->idMessage,NULL,0); //  Informa il mittente che i caratteri sono stati copiati
            break;                            // nel buffer. Il puntatore m->p punta ad un'area della
                                              // memoria comune a tutti i processi ed è stata
                                              // implementata in Linux per mezzo dei segmenti condivisi.
        case OP_WRITE:                        //  Questa porzione si occupa della trasmissione di caratteri
            break;                            // attraverso la linea seriale.
                 }
}
```

Appendice A10. Codice sorgente: magnetometer.h

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.magnetometer.h
    Created by Antonino Sicali
*/

#ifndef _MAGNETOMETER_H
#define _MAGNETOMETER_H

#include "osgm.h"

#define NMESSAGE                1      //  Indica il numero di messaggi che l'applicazione registrerà
#define MeasureSize             30     //  Indica la lunghezza di ciascuna misura magnetica
#define MaxNumberOfMeasure      10     //  Indica il numero massimo di misure memorizzabili nel buffer
#define MEASUREMENTS_BUFFER_SIZE 32000 //  Indica la dimensione in byte del buffer in cui vengono conservate
                                       // le misure

#endif  _MAGNETOMETER_H
```

Appendice A11. Codice sorgente: magnetometer.c

```
/*
    Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
    OSGM.magnetometer.c
    Created by Antonino Sicali
*/

#include "osgm.h"
#include "magnetometer.h"
#include "memory.h"

//  Nome di identificazione del processo nel sistema
char APPLICATION[]="Magnetometer";

//  Prima funzione chiamata all'avvio del processo. Ha le stessa semantica di main per i
// processi C/C++.
int osgmMain()
{
    int idSerialMessage;                    //  Questa varibile contiene l'identificativo del messaggio che
    int numberOfMeasures=MaxNumberOfMeasure;// permette di comunicare con periferiche esterne e connesse
    MessageHandleStruct messages[NMESSAGE]; // attraverso la linea seriale. La variabile 'numberOfMeasures'
    char *buffer;                           // contiene in ogni instante il numero di misure che è possibile
    IoQuery queryMeasure;                   // memorizzare ancora nel 'buffer'. La struttura 'queryMeasure'
                                            // permette di acquisire dalla linea seriale una singola misura.
                                            //  Naturalmente si può anche acquisirne un numero di misure
                                            // superiore.

    queryMeasure.size=MeasureSize;          //  Indica che si vuole leggere una singola misura dal buffer della
    queryMeasure.op=OP_READ;                // seriale.
    messages[0].id=registerMessage(APPLICATION); //  Un processo per ricevere indietro una richiesta deve
    messages[0].func=NULL;                      // necessariamente registrare un messagio nel sistema.
    queryMeasure.idMessage=messages[0].id;
    idApplication=messages[0].id;

    idSerialMessage=decodeMessage(SERIAL);      //  Acquisisce il numero di messaggio per comunicare con la
                                                // porta seriale.

    buffer=(char *) allocMemory(MEASUREMENTS_BUFFER_SIZE); //  Alloca nella memoria dinamica, comune a tutti
                                                           // i processi, un buffer di lettura/scrittura delle
                                                           // misure magnetiche.
    while(1) {
        //  Sposta il puntatore in un area libera della memoria allocata precedentemente
        queryMeasure.p=buffer+(MaxNumberOfMeasure-numberOfMeasures)*MeasureSize;
        //  Richiede la lettura di una misura dalla porta seriale. Intrinsecamente la funzione
        // queryService è bloccante.
        queryService(idSerialMessage,&queryMeasure,sizeof(IoQuery),NULL);
        //  Descrementa il numero di misure che è possibile scrivere nel buffer
        numberOfMeasures--;
        //  Se il buffer è pieno viene inoltrata una richiesta di scrittura nell'area a
        // permanente (FileSystem).
        if (!numberOfMeasures) {
                        //  Scrive le misure nell'area permanente (FileSystem)
                        writeBlock(buffer,MaxNumberOfMeasure*MeasureSize);
                        //  Libera il buffer di lettura
                        numberOfMeasures=MaxNumberOfMeasure;
                              }
         }
    return 0;
}
```

Appendice A12. Codice sorgente: Makefile

```
#/*
#           Istituto Nazionale di Geofisica e Vulcanologia - Osservatorio Etneo
#           OSGM.makefile
#           Created by Antonino Sicali
#*/

XLIB=/usr/X11R6/lib
XINCLUDE=/usr/X11R6/include
CC=cc

all: kernel memory io magnetometer athena

kernel: kernel.o klinux.o semaphore.o
           $(CC) -o kernel -Xlinker kernel.o klinux.o semaphore.o -lpthread

memory: linux.o memory.o semaphore.o
           $(CC) -o memory -Xlinker memory.o linux.o semaphore.o

magnetometer: linux.o magnetometer.o
           $(CC) -o magnetometer -Xlinker magnetometer.o linux.o

io: linux.o io.o
           $(CC) -o io -Xlinker io.o linux.o

motif: memory.h kernel.h osgm.h xosgm.h xosgm.c semaphore.o
           $(CC) $(OPTIONS) -D__SIMULATOR__ -D__MOTIF__ -o xosgm xosgm.c -Xlinker semaphore.o \
              -L$(XLIB) -I$(XINCLUDE) -lXm -lX11 -lXmu -lpthread

athena: memory.h kernel.h osgm.h xosgm.h xosgm.c semaphore.o
           $(CC) $(OPTIONS) -D__SIMULATOR__ -D__ATHENA__ -o xosgm xosgm.c -Xlinker semaphore.o \
              -L$(XLIB) -lXaw -lX11 -lXmu -lpthread

linux.o: kernel.h osgm.h linux.c
           $(CC) $(OPTIONS) -D__SIMULATOR__ -c linux.c

klinux.o: kernel.h osgm.h linux.c
           $(CC) $(OPTIONS) -D__SIMULATOR__ -D__KERNEL__ -c linux.c -o klinux.o

kernel.o: kernel.h osgm.h kernel.c
           $(CC) $(OPTIONS) -D__SIMULATOR__ -D__KERNEL__ -c kernel.c

memory.o: memory.h osgm.h memory.c
           $(CC) $(OPTIONS) -D__SIMULATOR__ -c memory.c

magnetometer.o: magnetometer.h osgm.h magnetometer.c
           $(CC) $(OPTIONS) -D__SIMULATOR__ -c magnetometer.c

semaphore.o: semaphore.c
           $(CC) $(OPTIONS) -c semaphore.c

io.o: io.h osgm.h io.c
           $(CC) $(OPTIONS) -D__SIMULATOR__ -c io.c

clean:
           rm semaphore.o
           rm klinux.o
           rm linux.o
           rm memory.o
           rm kernel.o
           rm magnetometer.o
           rm io.o
           rm xosgm
           rm kernel
           rm memory
           rm io
           rm magnetometer
```

ISSN 1590-2595

Quaderni di Geofisica

http://istituto.ingv.it/it/le-collane-editoriali-ingv/quaderni-di-geofisica.html

I Quaderni di Geofisica coprono tutti i campi disciplinari sviluppati all'interno dell'INGV, dando particolare risalto alla pubblicazione di dati, misure, osservazioni e loro elaborazioni anche preliminari, che per tipologia e dettaglio necessitano di una rapida diffusione nella comunità scientifica nazionale ed internazionale. La pubblicazione on-line fornisce accesso immediato a tutti i possibili utenti. L'Editorial Board multidisciplinare garantisce i requisiti di qualità per la pubblicazione dei contributi.

ISSN 2039-7941

rapporti tecnici INGV

http://istituto.ingv.it/it/le-collane-editoriali-ingv/rapporti-tecnici-ingv.html

I Rapporti Tecnici INGV pubblicano contributi, sia in italiano che in inglese, di tipo tecnologico e di rilevante interesse tecnico-scientifico per gli ambiti disciplinari propri dell'INGV. La collana Rapporti Tecnici INGV pubblica esclusivamente on-line per garantire agli autori rapidità di diffusione e agli utenti accesso immediato ai dati pubblicati. L'Editorial Board multidisciplinare garantisce i requisiti di qualità per la pubblicazione dei contributi.

ISSN 2039-6651

Miscellanea INGV

http://istituto.ingv.it/it/le-collane-editoriali-ingv/miscellanea-ingv.html

La collana Miscellanea INGV nasce con l'intento di favorire la pubblicazione di contributi scientifici riguardanti le attività svolte dall'INGV (sismologia, vulcanologia, geologia, geomagnetismo, geochimica, aeronomia e innovazione tecnologica). In particolare, la collana Miscellanea INGV raccoglie reports di progetti scientifici, proceedings di convegni, manuali, monografie di rilevante interesse, raccolte di articoli, ecc.

Coordinamento editoriale e impaginazione
Centro Editoriale Nazionale | INGV

Progetto grafico e redazionale
Daniela Riposati | Laboratorio Grafica e Immagini | INGV

Via di Vigna Murata, 605
00143 Roma
Tel. +39 06518601 Fax +39 065041181

http://www.ingv.it

Istituto Nazionale di Geofisica e Vulcanologia