UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Facoltà di Ingegneria – Sede di Modena

Corso di Laurea in Ingegneria Informatica

UNIX FUNCTION HELPER: SUPPORTO ALL'APPRENDIMENTO DELLE PRIMITIVE UNIX

Relatore

Chiar.mo Prof. Letizia Leonardi

Elaborato di

Marco Sentimenti

Anno Accademico 2011-2012

Indice

Introduzione

In questo elaborato viene trattata la realizzazione dell'applicazione Unix Function Helper, programma atto a fornire un elemento di supporto alla didattica per l'apprendimento delle caratteristiche di alcune primitive del sistema Unix.
L'elaborato è strutturato in 5 capitoli che affrontano 5 tematiche differenti relative all'applicazione suddetta.
Il primo capitolo offre una panoramica generale sul progetto, giustificando i motivi che hanno determinato le principali scelte implementative, e indica alcuni requisiti base per un'esecuzione ottimale.
Il secondo capitolo spiega la tipologia di file nel quale sono state compresse tutte le risorse utilizzate dal programma e come queste vengano utilizzate dal programma stesso.
Il terzo capitolo mostra la tematica che fa da sfondo all'intero progetto: vengono elencati, infatti, elementi del disegno bidimensionale con diversi esempi dal punto di vista teorico confrontati con estratti dei sorgenti dell'applicazione.
Il quarto capitolo presenta la parte essenziale dell'implementazione, in particolare quella basata sull'uso dei thread e su come thread e grafica siano strettamente correlati.
Infine il capitolo quinto analizza, passo dopo passo, tutte le opzioni di cui l'utente finale può disporre all'interno del programma

1. Descrizione del progetto

Il progetto si è basato sulla realizzazione di un'applicazione che potesse garantire un supporto per la didattica agli studenti relativamente ad alcune primitive presentate nelle lezioni di Sistemi Operativi e Lab. In particolare si è cercato di focalizzare l'attenzione sulla visualizzazione grafica degli effetti di alcune primitive del sistema Unix. Una primitiva è un'azione elementare eseguita senza interruzione dal sistema operativo, ma visibile all'interno di un linguaggio ad alto livello (ad esempio il C) come una normale procedura. Le primitive selezionate per questo programma sono collegate alle operazioni sui processi e sui file,
in particolare le primitive Fork, Exec, Open e Creat.
La primitiva Fork è in grado di generare un nuovo processo il quale acquisisce, per copia, le informazioni del processo che invoca la primitiva.
La primitiva Exec cambia il codice del processo chiamante con il codice un file eseguibile, ovvero permette di eseguire un programma specifico in sostituzione al programma che si sta eseguendo all'interno del processo. Il file deve essere specificato come parametro.
La primitiva Open permette di aprire un file residente nella memoria di massa del sistema. È l'operazione preliminare da eseguire da parte del processo chiamante per poter interagire con il file secondo la modalità di apertura specificata.
La primitiva Creat, infine, genera un nuovo file e lo apre in modalità scrittura per il processo che esegue tale primitiva. È possibile invocare la Creat anche su file preesistenti qualora si voglia cancellarne il contenuto.
Per poter facilitare l'apprendimento di queste nozioni, si è pensato di costruire un'interfaccia grafica nella quale lo studente potesse osservare e combinare tutte le opzioni possibili a proprio piacimento: in questo modo anche alcuni aspetti che normalmente vengono trascurati possono essere colti molto velocemente, dal momento che ogni elemento è accompagnato dalla relativa descrizione e spiegazione.
Una problematica affrontata riguarda sicuramente la portabilità del programma. Infatti essendo un software che deve essere disponibile per tutti gli studenti, deve essere allo stesso tempo utilizzabile da qualunque macchina posseduta dagli stessi; in particolare non ci devono essere problemi dal punto di vista del sistema operativo.
Combinando quindi la necessità di un'applicazione portabile che, allo stesso tempo, garantisse un'interfaccia grafica intuitiva, si è deciso di scrivere l'applicazione nel linguaggio di programmazione Java.
La caratteristica principale del linguaggio Java è proprio la portabilità, poiché può essere eseguito indipendentemente dalla piattaforma hardware o software in cui si trova. Il codice sorgente infatti non è direttamente trasformato in codice binario per essere elaborato da un processore, ma viene compilato in un linguaggio intermedio, il bytecode. Il bytecode è analogo al linguaggio macchina, ma deve essere interpretato da una macchina virtuale scritta per l'hardware sottostante per poter essere eseguito. È possibile quindi compilare un programma Java in ambiente Windows ed eseguirlo in ambiente Linux.
Se l'utente dispone dunque della JVM (Java Virtual Machine) per il proprio sistema, è in grado di eseguire qualunque programma Java a meno di limitazioni dovute alla versione del linguaggio. Non conoscendo le preferenze hardware e software degli studenti, è possibile soddisfarne la totalità tramite l'utilizzo di suddetto linguaggio.
Per quanto riguarda l'implementazione effettiva, il progetto è stato realizzato tramite l'IDE (Integrated Development Environment) Eclipse Juno 4.2 e sfruttando la JDK (Java Development Kit) 6 update 31.
Un normale utente utilizza una JRE (Java Runtime Environment) presente sul proprio sistema per eseguire normali applicazioni Java standalone, oppure nei browser web per eseguire delle applet.
Non sono presenti differenze tra JDK e JRE dal punto di vista dell'utente, mentre il programmatore predilige la JDK. La JRE è un'implementazione della JVM specifica per un sistema operativo ed un'architettura hardware. Oltre alla JVM, include librerie di base e alcuni componenti aggiuntivi per

eseguire le applicazioni e le applet scritte in Java. La JDK è un'estensione della JRE e a questa aggiunge il software necessario per sviluppare le applicazioni Java: la parte software aggiuntiva più importante riguarda il compilatore Java, che genera il codice bytecode dai file sorgenti, e l'insieme delle API (Application Programming Interface), ovvero le procedure disponibili al programmatore.
La JDK ovviamente richiede più spazio su disco, in quanto presenta molte più componenti della JRE.
Si consiglia per la corretta esecuzione del programma la presenza della JDK o la JRE 6 (o successive) sul proprio sistema. Per quanto riguarda la versione dell'update non ci sono particolari requisiti se non per la visualizzazione di un'interfaccia, o più propriamente un “LookAndFeel”, diversa da quella utilizzata di default dai programmi Java: il “LookAndFeel” in questione è chiamato Nimbus ed è disponibile solo dall'update 10.

2. Struttura dell'implementazione

2.1 Introduzione sul file Jar

Un file Jar (Java Archive) è un archivio basato sull'algoritmo ZIP che permette di comprimere tipologie di risorse differenti in un unico file. Tutte le risorse sono legate ad un unico file Jar. Essendo un archivio, può essere utilizzato come una libreria: infatti è possibile richiamare file Jar da altri file Jar all'interno del proprio progetto. Per poter creare un file Jar è possibile utilizzare un'utility fornita direttamente dalla JDK (Java Development Kit) ovvero “jar” con le opzioni “c” ed “f” per indicare, come output del comando, il file che deve essere generato.

Esempio. *jar cf jarFile InputFiles*

Gli InputFiles devono essere già stati compilati ovvero devono presentare l'estensione .class.
Ad un file Jar viene sempre associato un file denominato MANIFEST.MF, il quale contiene le informazioni che identificano quel particolare Jar a livello di contenuti, come per esempio le indicazioni sul suo ClassPath.
All'interno di questo file le informazioni, devo essere indicate secondo il formato “nome: valore”.
Se il Jar è eseguibile occorre indicare la classe Main dell'applicazione.
Alla creazione del Jar viene automaticamente inserito un Manifest di default; per creare il proprio Manifest occorre utilizzare l'opzione “m” all'interno del comando “jar”.

Esempio. *jar cmf manifestFile jarFileName inputFiles*

Da Eclipse, tramite l'opzione *Export → Runnable Jar* File, il Manifest viene costruito all’interno del Jar di destinazione in modo da contenere già tutte le informazioni necessarie all’applicazione, e tutte le classi del progetto selezionato entrano a far parte del Jar (a meno di direttive specifiche da parte del programmatore).

2.2 Struttura File Jar e nota all'esecuzione in ambiente Linux

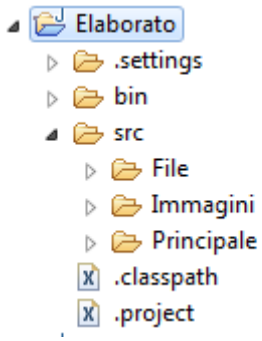

Figura 2.1 Struttura del Jar

Come mostrato in Figura 2.1, l'applicazione comprende un unico file Jar, il quale è costituito da 3 package:

1) il package con le classi Java, denominato Principale;
2) il package con il file “.png” utilizzati dal programma, denominato Immagini;
3) il package con i file di testo contenenti le spiegazioni sulle primitive, denominato File.

Per poter eseguire il file Jar in ambiente Linux è necessario qualche passaggio preliminare:

1) aprire un terminale ed posizionarsi nella cartella in cui è presente il file Jar;
2) dare il comando *chmod u+x Unix Function Helper*, per rendere il file eseguibile;
3) lanciare il Jar digitando *java -jar Unix Function Helper*.

In alternativa al punto 3), per non eseguire il jar ogni volta da terminale, si può impostare come programma predefinito per l'apertura di quel tipo di file la piattaforma Java installata nel proprio pc. Questo consente il lancio dell'applicazione con un semplice doppio click del mouse, in modo da uniformarsi agli altri sistemi operativi.

2.3 Reperire le risorse del Jar a livello di codice

Il meccanismo con cui la JVM (Java Virtual Machine) ricerca le classi da caricare si basa sull'esaminare il ClassPath. Il CLASSPATH è una variabile d'ambiente necessaria al compilatore per specificare la locazione in cui posizionare le classi compilate, e fondamentale a runtime per poterle caricare correttamente: contiene una lista di directory, separate da ":" o ";" a seconda del sistema operativo in uso nella quali i comandi java e javac cercano automaticamente i file .class; un pò come succede per la ricerca dei comandi da eseguire con la variabile PATH.
In Eclipse, i file sorgente e quelli compilati vengono generalmente posizionati in 2 sottocartelle distinte della stessa directory (quella del progetto).
Il classpath, che come valore di default indica la directory "." , può essere modificato in vari modi:

1) agendo sulla variabile d'ambiente tramite interfaccia grafica e, a seconda del sistema operativo, controllare di inserire il giusto carattere separatore. Queste sono modifiche permanenti alla variabile d'ambiente corrispondente;
2) da linea di comando (il comando può variare nei vari sistemi operativi) come per esempio SET CLASSPATH per Windows o export CLASSPATH per Linux (dopo aver opportunamente definito CLASSPATH); questa possibilità consente di settare il classpath per la sessione corrente;
3) tramite l'opzione -classpath (o -cp) offerta dalla JVM al momento della compilazione o del lancio del programma. È necessario specificare la gerarchia di package, se presente, per indicare la classe o il file sorgente da utilizzare. Al momento del lancio dell'applicazione occorre ovviamente dare come parametro il file compilato che contiene il metodo main.

Esempio. *javac -classpath direttorio_della _risorsa package.fileSorgente*
java -classpath direttorio_della _risorsa package.fileCompilato

È importante sottolineare che ogni modo sopracitato, quando viene utilizzato, annulla il settaggio precedente al momento della ricerca delle risorse.

La JVM si serve poi di una classe astratta denominata ClassLoader, che segue le direttive della variabile CLASSPATH, o degli altri metodi sopracitati, per determinare la posizione dei file compilati.
Dal punto di vista del Jar la situazione non è molto differente: poichè il file Jar è un'estensione del formato Zip, implementa alcuni funzionalità proprie di un filesystem, ovvero la possibilità di specificare ed utilizzare direttori, file e percorsi relativi o assoluti; la radice di questo filesystem è il jar stesso. Il Jar posside un proprio ClassPath, indicato nel Manifest, ed è attraverso quel ClassPath che il ClassLoader ricerca le risorse nel Jar stesso. Aggiungendo dunque un Jar al ClassPath di una applicazione, il meccanismo per estrarne le classi è identico all'esplorazione di un normale direttorio da parte del ClassLoader. Per le risorse che non siano classi, l'applicazione java deve

invece indicarne il corretto percorso all'interno del codice, sfruttando i metodi *getResouceAsStream()* o *getResource()* delle classi ClassLoader/Class per poi utilizzarne i dati nel modo più oppurtuno. La differenza tra i 2 metodi sta principalmente nel tipo di dato che ritornano: *getResource()* ritorna un URL, mentre *getResourceAsStream()* un InputStream, ed entrambi richiedono come parametro una stringa che indichi il percorso riguardante la risorsa da selezionare. Non c'è una suddivisione precisa delle tipologie di risorse da utilizzare con un metodo piuttosto che con l'altro poichè dipende da applicazione ad applicazione: per esempio si potrebbe utilizzare *getResourceAsStream()* per localizzare un file di testo e disporre già del relativo InputStream per interagire con il file stesso, oppure utilizzare il metodo *getResource()* e sfruttare il metodo *openStream()* della classe URL. La differenza tra ClassLoader e Class invece dipende da una differente impostazione della ricerca all'interno del jar: con ClassLoader la directory di partenza è la root del file jar quindi occorre dare in input un percorso a partire dalle sottodirectory della root, mentre con Class se la root non è inserita manualmente nel percorso si presuppone che la risorsa sia nello stesso package della classe Class. Prendendo come esempio il Jar dell'applicazione realizzata, in particolare focalizzando l'attenzione sui componenti evidenziati dalla Figura 2.2:

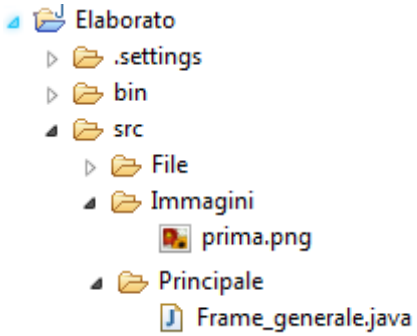

Figura 2.2 Particolare del Jar

Se la classe Frame_generale necessita della risorsa "prima.png", contenuta nella sottodirectory di root Immagini, può dunque percorrere 2 strade:

1) utilizzare il metodo getSystemClassLoader() della classe astratta ClassLoader in questo modo

ClassLoader.getSystemClassLoader().getResourceAsStream("Immagini/prima.png");

oppure

ClassLoader.getSystemClassLoader().getResource("Immagini/prima.png");

2) utilizzare il metodo getClass della classe Class

getClass().getResourceAsStream("/Immagini/prima.png");

oppure

getClass().getResource("/Immagini/prima.png");

facendo attenzione all'inserimento del carattere "/" all'inizio della stringa fornita come parametro. Se prova.png venisse posizionata nel package Principale, potrebbe essere caricata tramite l'istruzione *getClass().getResource("prova.png").*

Una piccola nota sul metodo *getSystemClassLoader()*. Poichè i metodi *getResource()* e *getResourceAsStream()* ricevono in ingresso una stringa da considerare come un percorso, e poichè dunque inserendo nel percorso “immagine.png” si ottiene ./immagine.png (ovvero il riferimento alla root del file Jar), all’interno del percorso specificato non deve mai comparire il carattere “/” iniziale se si sta utilizzando ClassLoader altrimenti la ricerca non produrrà alcun risultato utile. Il metodo *getSystemClassLoader()* è utilizzato soprattutto in contesti statici quando non c'è un oggetto this a cui riferirsi, mentre in contesti non statici viene utilizzato *getClass()*.

Per fornire un nome relativo di una directory non presente nella root del Jar occorre aggiungere una nuova entry nel Classpath del file Manifest. Nel file Manifest infatti, l'indicazione di partenza del ClassPath è . , quindi il ClassLoader capisce che gli elementi da caricare saranno nella root del Jar (a meno di percorsi assoluti specificati al momento del caricamento). Dunque per consentire al ClassLoader di caricare risorse presenti in directory differenti si opera nel seguente modo: si inseriscono le directory necessarie nel file Manifest alla voce Classpath con l’accortezza di utilizzare i caratteri “.”e “/” all’inizio di ogni percorso, per indicare la root del Jar come punto di partenza, e di porre lo “/” finale al percorso inserito (per indicare che la ricerca va effettuata all’interno di quella cartella). Supponiamo per esempio di volere creare una cartella Risorse all’interno del Jar, contenente le cartelle Immagini e File. Per permettere al sistema di caricare correttamente le risorse in queste 2 cartelle occorre accedere al file Manifest e inserire la seguente scrittura

Class-Path: /Risorse/

3. Grafica dei pannelli

3.1 Introduzione

Le librerie standard di Java mettono a disposizione nel package java.awt un insieme di classi per la grafica bidimensionale. Tra le innumerevoli classi presenti occorre focalizzare l'attenzione su Graphics e Graphics2D. La classe Graphics è una classe astratta che permette di richiamare tutti metodi legati al disegno bidimensionale in Java. Quando infatti il programmatore si appresta a disegnare tramite queste librerie su un opportuno container (generalmente un componente di AWT o di Swing), non deve fare altro che eseguire l'override di un metodo particolare, il metodo *paint(Graphics g)*: infatti l'oggetto g di tipo Graphics, che questo metodo riceve in input, rappresenta il contesto grafico corrente (è formato infatti da diversi attributi che determinano colore, tipologia del Font, etc...) e permette di richiamare i metodi necessari ad ogni tipologia di operazione grafica. Una particolarità di *paint()* riguarda la sua invocazione, che è automatica e delegata all'applicazione tutte le volte che è necessario un nuovo aggiornamento della grafica del componente in considerazione, per esempio a causa dello scrolling della finestra, o anche semplicemente quando esso viene visualizzato. Il programmatore però può richiedere esplicitamente di ridisegnare il componente attraverso il metodo *repaint()*, il quale porta alla chiamata di un metodo *update()* che cancella la superficie del disegno e richiama il metodo *paint()* del componente stesso. Questa possibilità conferita dalla funzione *repaint()* è stata utilizzata molte volta all'interno dell'applicazione sopracitata, per far si che il disegno si modelli a seconda delle istruzioni impartite dall'utente attraverso l'uso dei bottoni presenti nell'applicazione stessa.
A partire dalla versione 2 di Java, la classe Graphics è stata estesa dalla classe Graphics2D, la quale aggiunge molteplici funzionalità alla libreria già ricca del package AWT. Questa classe permette infatti di avere un controllo più sofisticato sulle geometrie del disegno, come nelle trasformazioni di coordinate, nel layout del testo e nella gestione del colore.
Un esempio evidente, utilizzato anche all'interno dell'applicazione sopracitata, riguarda il controllo della qualità del rendering della grafica:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,RenderingHints.VALUE_ANTIALIAS_ON);
```

il metodo *setRenderingHint()* permette, in particolare, di modificare un singolo aspetto del rendering a seconda della coppia di parametri che viene scelta dal programmatore; il primo elemento deve essere l'ambito della grafica su cui intervenire, il secondo elemento è il valore che si desidera utilizzare (entrambi i valori sono definiti nella classe RenderingHints).
L'ambito utilizzato nell'esempio è quello dell'antialiasing, il quale migliora la qualità del rendering delle linee oblique ostacolando il fenomeno dell'aliasing.
Supponiamo infatti di trovarci nella situazione mostrata nelle Figure 4.1 e 4.2. Poichè un'immagine digitale è formata da una griglia di pixel, essa è in grado di rappresentare le linee oblique solo creando un effetto frastagliato (Figura 4.1), dal momento che i pixel sono quadrati. Questo effetto si nota tanto più quanto è forte la differenza di colore tra la linea e l'area circostante.
L'antialiasing permette di conferire una colorazione intermedia tra il colore della linea e quello dell'area circostante ai pixel direttamente adiacenti a quelli che creano l'effetto non voluto, rendendo più graduale il passaggio da un colore all'altro (Figura 4.2).

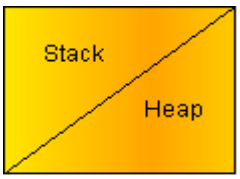

Figura 4.1 Rendering senza Antialiasing

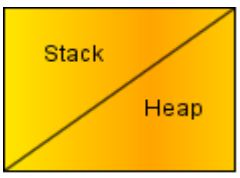

Figura 4.2 Rendering con Antialiasing

Dal momento che l'antialiasing porta molti più calcoli per il motore grafico al momento del rendering, esso può influire sulle prestazioni generali dell'intera applicazione. (non è ovviamente il caso dell'applicazione presa in considerazione) In generale tutte le opzioni del metodo *RenderingHint()* costituiscono un trade-off tra velocità dell'applicazione e qualità del disegno.

Per poter usufruire delle potenzialità aggiuntive offerte dalla classe Graphics2D occorre eseguire il seguente casting esplicito nel metodo paint().
g2 = (Graphics2D) g dove g è il parametro di tipo Graphics in input al metodo *paint()*.

3.2 Elementi grafici utilizzati all'interno dell'applicazione

3.2.1 Sistema di coordinate

Il sistema di coordinate in Java reso disponibile al programmatore per il disegno bidimensionale è chiamato User-space ed è indipendente dallo strumento di output utilizzato per mostrare il risultato del rendering: si parla infatti di User-space e Device-space. Le coordinate del Device-space variano invece a seconda dell'oggetto su cui viene eseguito il rendering; a volte sono molto diversi tra loro ma queste differenze sono trasparenti al programma e al programmatore.
Tutti le geometrie espresse secondo i metodi di Java 2D sono specificate in questo spazio-utente. Le coordinate user vengono trasformate nelle coordinate device quando il componente deve essere renderizzato (ovvero quando ne viene invocato il metodo *paint()*) e questo tipo di conversione avviene in modo completamente automatico.
L'angolo superiore sinistro ha coordinate (0, 0) e gli assi x e y si prolungano rispettivamente verso destra e verso il basso. Le coordinate sono specificate solitamente come interi (infatti molti metodi richiedono parametri interi, ad esempio quelli per posizionare le figure geometriche), ma per una maggiore precisione del disegno sono supportate anche le coordinate di tipo float e double. La scelta effettuata in questa applicazione è legata all'uso delle coordinate di tipo double con conseguenti conversioni esplicite ad interi per i metodi che lo richiedono.

3.2.2 Metodi di Rendering

Come accennato nell'introduzione del capitolo, il disegno tramite la classe Graphics (o Graphics 2D) si basa principalmente sullo sfruttamento di un oggetto di suddetta classe che mantiene salvato il contesto grafico secondo le impostazioni dettate dal programmatore, o secondo quelle presenti di default. Il contesto desiderato può essere modificato in qualunque momento richiamando i metodi opportuni e permette di disegnare gli elementi tipici della grafica bidimensionale, sfruttando i cosiddetti metodi di rendering. I principali metodi che ricadono in questa categoria comportano la rappresentazione di una figura geometrica, il riempimento della sua area, la renderizzazione del testo, o il disegno di un'immagine.
Per quanto riguarda la rappresentazione delle figure geometriche occorre fare distinzione tra la classe Graphics e la Graphics2D: infatti la prima presenta un metodo di contorno e di riempimento per ogni figura possibile e le disegna sfruttando coordinate intere, mentre la seconda dispone di un singolo metodo per perimetro e area che prende in ingresso l'istanza di una classe che implementa l'interfaccia Shape (interfaccia che comprende un elemento, chiamato PathIterator, che contiene le regole per definire i punti all'esterno o all'interno della figura geometrica, nonché l'espressione del suo perimetro). Per Graphics2D inoltre la rappresentazione avviene secondo coordinate float o double quindi con maggiore precisione all'interno dell'applicazione rispetto alla classe Graphics

Esempio. *g.drawLine((0,(int) centroY, (int)d.width, (int)centroY);*

g2.draw(new Line2D.Double(0, centroY, d.width, centroY));

La prima istruzione sfrutta l'oggetto g di Graphics e disegna una linea
La seconda, partendo dall'oggetto g2 di Graphics2D, realizza una linea in coordinate a virgola mobile e doppia precisione. Entrambi i metodi ricevono in input il punto di partenza e il punto di arrivo della linea stessa. CentroY e d.width sono misure legate alla dimensione del Frame principale, in particolare centroY è la metà della sua altezza.

Esempio. *g.fillRect((int)(d.width - DIST_BORDO_DESTRO - ud.getLARGHEZZA_COD()), (int)(d.height- ud.getALTEZZA_TAB() - DIST_BORDO_INFERIORE), (int)(ud.getLARGHEZZA_COD()),(int)(ud.getALTEZZA_COD()));*

g2.fill(new Rectangle2D.Double(d.width - DIST_BORDO_DESTRO - ud.getLARGHEZZA_COD(), d.height- ud.getALTEZZA_TAB() - DIST_BORDO_INFERIORE, ud.getLARGHEZZA_COD(), ud.getALTEZZA_COD()));

L'esempio mostra le differenze tra i metodi *fill()* di Graphics e Graphics2D
Sfruttando sempre g2, il metodi *fill()* si occupa di riempire l'area della figura che riceve in input con il colore del contesto grafico corrente. I valori in ingresso riguardano il punto di partenza del rettangolo (il vertice in alto a sinistra), la larghezza, e l'altezza. Questo metodo è utilizzato per colorare l'area di codice all'interno delle primitive Fork ed Exec.

Il metodo per renderizzare il testo è *drawString()* ed è il medesimo per entrambe le classi Graphics e richiede delle coordinate intere.
Per quanto riguarda le immagini, nonostante queste classi presentino diversi metodi *drawImage()*, si è preferito ricorrere alla classe ImageIcon ed associare ogni immagine direttamente ad un bottone, dal momento che sono state utilizzate solo delle piccole icone.

3.2.3 Color e Stroke

Con il metodo *fill(),* la classe Graphics2D permette di eseguire il riempimento dell'area di una determinata figura geometrica secondo il colore del contesto grafico. I colori in Java seguono il modello RGB (Red,Green,Blue), implementato nella classe Color: infatti, a parte le costanti relative ai colori principali, è possibile eseguire qualunque combinazione di colori inserendo un numero intero compreso tra 0 e 255 nei 3 campi del costruttore di suddetta classe (i campi diventano 4 se si vuole controllare anche l'opacità del colore).

Esempio.

Color c = new Color(255,165,0); definisce una tonalità di arancione

Color c1 = new Color(0,0,0,200); definisce una tonalità di nero molto opaca

Per impostare il colore del contesto grafico si utilizza il metodo setPaint() di Graphics2D o Graphics.

Con l'introduzione della classe Graphics2D il programmatore può inserire dei gradienti di colore, aumentando così le combinazioni possibili di colorazioni da conferire ai vari componenti.
Per conferire un riempimento a gradiente occorre settare il colore definendo un oggetto della classe GradientPaint.

Esempio.

GradientPaint gr = new GradientPaint(0,0,Color.lightGray,100, 0,new Color(173,216,230),true);

In questo costruttore sono richiesti i 2 colori che devono comporre il gradiente ed occorre anche indicare se si vuole realizzare un gradiente ciclico o aciclico, ovvero ripetere ciclicamente i 2 colori o mantenerli distinti nel momento in cui si passa dal primo al secondo. La partenza del primo colore e il passaggio dal primo al secondo sono definite dai 2 punti ricevuti in input dal costruttore.

Un altro elemento importante nella grafica bidimensionale riguarda il rendering delle linee, o meglio del perimetro di una qualsiasi figura: infatti, tramite la classe BasicStroke, è possibile intervenire su diversi aspetti, come ad esempio la dimensione del tratto, lo stile, etc.. In particolare, all'interno dell'applicazione si possono trovare esempi di utilizzo di metodi per variare la tipologia del tratto. La classe BasicStroke è stata introdotta sempre con l'estensione Graphics2D, della quale va a modificare l'attributo Stroke.

Esempio.

```
float dash1[] = {10.0f};
BasicStroke dashed = new BasicStroke(1.0f,BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER, 10.0f, dash1, 0.0f);
g2.setStroke(dashed);
```

Il parametro dash1 definisce la tipologia della linea (continua o trattegiata) e quanta distanza devono mantenere i vari tratti della linea. Gli altri valori numerici riguardano lo spessore e la nitidezza del tratto. I valori costanti invocati direttamente dalla classe BasicStroke definiscono invece il comportamento che la linea deve osservare nel momento in cui avvengono delle intersezioni tra 2 tratti, oppure le decorazioni da porre al termine del singolo tratto della linea.

3.2.4 Caratteristiche del Testo

Come già introdotto nel sottoparagrafo 3.2.2, una stringa di testo viene renderizzata tramite il metodo *drawString()*: in ingresso il metodo riceve il punto dal quale iniziare a scrivere la prima lettera, e poi ovviamente la stringa da visualizzare. Dal punto di vista del sistema di coordinate, la renderizzazione della stringa avviene verso l'alto e verso destra rispetto al punto fornito come parametro.
Per disegnare del testo si fa riferimento ad uno specifico Font. Un font è un insieme di caratteri e/o di simboli di forma omogenea al quale è possibile applicare diverse operazioni o, per meglio dire, è un collezione di glyphs che renderizzano un preciso set di caratteri. Se infatti un carattere è la rappresentazione di una lettera o di un certo simbolo, un glyph è una figura utilizzata per renderizzare un carattere o una sequenza di caratteri.
La classe di riferimento è Font, introdotta già dalle prime versioni di Java, il cui costruttore è il seguente:

Font f = new Font(font.getFontName(), Font.PLAIN, (int) dimensione_font);

Il nome del font identifica chiaramente la tipologia di caratteri a cui stiamo facendo riferimento. In questo esempio è stato preso il nome del font del contesto grafico corrente, tramite l'istruzione
Font font= g2.getFont();
I font possono essere classificati in 2 tipologie, font fisici e font logici: i primi identificano un preciso font all'interno del sistema e quindi un particolare tipo di font fisico può non esistere su un'altra macchina; i secondi sono 5 famiglie di font definiti dall'ambiente Java e che trovano corrispondenza con i font fisici presenti nel sistema. I font logici garantiscono la portabilità dell'applicazione da questo punto di vista. Un esempio può essere dato dal font logico SansSerif, che corrisponde ad Arial su Windows e ad Helvetica su MacOS.
Il secondo parametro riguarda lo stile del font (PLAIN, BOLD, ITALIC), mentre il terzo riguarda la dimensione misurata in punti: per dimensione in punti si intende l'altezza dei caratteri di un font, ovvero la distanza tra il punto superiore del carattere più alto e il punto inferiore del carattere più basso. Un punto equivale a 1/72 di pollice.

3.3 Analisi della realizzazione di un pannello specifico

Nei paragrafi precedenti sono stati introdotti i principali elementi utilizzati dall'applicazione per il disegno dei pannelli che si alternano nella parte centrale del frame generale.
In questo paragrafo, si intende analizzare con un dettaglio maggiore un pannello particolare, evidenziandone la struttura di fondo e il meccanismo di funzionamento che, come per gli altri pannelli, permette la rappresentazione degli aspetti salienti delle primitive prese in esame.
Il pannello preso come esempio è quello relativo alla primitiva Open, anche se la struttura è completamente identica agli altri 3 pannelli, fatta eccezione per alcuni elementi caratterizzanti delle singole primitive (essendo primitive differenti, agiscono su elementi differenti del sistema e quindi necessitano di rappresentazioni differenti).
Nel Listato 3.1 viene riportata la parte principale del codice.

```
public void paint(Graphics g){

g2 = (Graphics2D)g;
//salvo le dimensioni del pannello perchè servono per i posizionamenti delle
varie figure
d=this.getSize();

[…]

//attivazione antialiasing per eliminare le linee oblique frastagliate
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,RenderingHints.VALUE_ANTIALI
AS_ON);

//disegno di partenza
if(parent_frame.getDisegna()==1||parent_frame.getDisegna()==2){
disegno_base();
}
//situazione dopo la primitiva
if(parent_frame.getDisegna()==2 && parent_frame.getSc().getStep()==0){

//descrittore del file nella tabella dei file aperti del processo
rect_text_table();
//rettangolo I/O pointer per la tabella dei file aperti di sistema
rect_IO_pointer();

//rettangolo copia i-node
rect_copia_inode();
if(parent_frame.getDisegnafrecce_padre()==1){
```

```
//puntatore chiamante-tabella file aperti di sistema
puntatore_proc_IO();

//freccia tra le tabelle di sistema
puntatore_tabelle();

//collegamento I/O pointer --- file
collegamento_IO_file();

//collegamento copia i-node --- i-node
collegamento_copia_inode();

   }

}


if(parent_frame.getSc().getStep()==1){
//step1 : descrittore del file
rect_text_table();
}

else if(parent_frame.getSc().getStep()==2){
//step1 : descrittore del file
rect_text_table();

//step 2: i/O pointer e puntatori
rect_IO_pointer();
//puntatore chiamante-tabella di sistema
puntatore_proc_IO();
//collegamento I/O pointer --- file
collegamento_IO_file();
}

else if(parent_frame.getSc().getStep()==3){

//step1 : descrittore del file
rect_text_table();

//step 2: i/O pointer e puntatori
rect_IO_pointer();
//puntatore chiamante-tabella di sistema
puntatore_proc_IO();
//collegamento I/O pointer --- file
collegamento_IO_file();

//step 3 : copia i-node e collegamenti rimanenti
//rettangolo copia i-node
rect_copia_inode();
//puntatore tra le tabelle di sistema
puntatore_tabelle();
//collegamento copia i-node --- i-node
collegamento_copia_inode();

}
```

Listato 3.1 Struttura di base del pannello Open

In questo pannello, come negli altri, il disegno delle varie figure geometriche è regolato da un gruppo di variabili appartenenti alla classe Frame_generale, la cui istanza è passata come parametro al costruttore. È semplicemente un insieme che regola la logica dell'applicazione, come il contesto grafico regola la logica del disegno. La struttura del codice risulta essere un esempio della gestione relativa ai metodi *paint()* ed *update()*: infatti essa corrisponde ad una serie di if che controllano le variabili sopracitate per indicare all'applicazione il disegno da eseguire; questo perchè il metodo *update()* esegue una cancellazione prima di chiamare il metodo *paint(),* quindi non si può contare sui disegni eseguiti precedentemente per effettuare quelli successivi. Dunque ogni blocco separato deve essere in grado di disegnare tutti i componenti relativi alla variabile che lo controlla.
Tra tutte queste variabili, quelle utilizzate per questo pannello sono:

- la variabile Disegna permette la rappresentazione della situazione presente prima della chiamata della primitiva se il suo valore è 1, mentre se il valore è 2 la situazione visualizzata sarà riferita a ciò che succede a causa dell'azione della primitiva;
- la variabile Disegna_frecce_padre garantisce la corretta visualizzazione dei puntatori che dal processo chiamante o processo padre, a seconda dei casi, raggiungono tutte le aree di interesse;
- la variabile Step infine è legata ad una modalità di visualizzazione graduale dell'azione della primitiva (si veda il capitolo 5 sulla spiegazione della GUI) e fondamentalmente definisce quale parte dell'immagine, che si avrebbe con Disegna=2, deve essere rappresentata. Se questa variabile è uguale a 0, significa che la visualizzazione graduale non è attivata e quindi si procede con la normale rappresentazione della situazione antecedente o successiva all'azione della primitiva.

Il valore di queste variabili, insieme a quelle utilizzate negli altri pannelli, vengono tutte manipolate dall'utente tramite la pressione dei bottoni dell'applicazione.
Ora analizziamo il pannello Open. La primitiva Open agisce su 3 differenti tabelle: la tabella dei file aperti del processo chiamante, la tabella dei file aperti di sistema e la tabella dei file attivi, nonché sulla memoria di massa. Il primo metodo che si incontra scorrendo il Listato 3.1 è quello del disegno di base, che deve preparare l'ambiente per l'azione della primitiva: esso deve quindi predisporre il disegno con le 3 tabelle più la memoria di massa sia nel caso in cui la variabile Disegna sia uguale 1, sia nel caso in cui sia uguale a 2.

```
//rettangolo per la tabella dei file aperti di sistema
g2.draw(new Rectangle2D.Double(centroX+DIST_CENTRO, d.height-
DIST_BORDO_INFERIORE-ud.getALTEZZA_TAB(), ud.getLARGHEZZA_TAB(),
ud.getALTEZZA_TAB()));

//rettangolo i-node presente nella memoria di massa
GradientPaint gr = new GradientPaint(0,0,Color.red,100, 0,new
Color(252,127,127),true);
g2.setPaint(gr);
g2.fill(new Rectangle2D.Double(d.width-DIST_BORDO_DESTRO-80,
DIST_BORDO_SUPERIORE+(ud.getALTEZZA_TAB()/5*2)+80, 80, 30));
g2.setColor(Color.black);
g2.drawString("I-Node", (int)(d.width-DIST_BORDO_DESTRO-
55),(int)(DIST_BORDO_SUPERIORE+(ud.getALTEZZA_TAB()/5*2)+100));
```

Listato 3.2 Estratto del metodo disegno_base

Nel metodo disegno_base (un estratto nel Listato 3.2), come nel resto del programma, si può notare più da vicino il discorso del contesto grafico corrente: ad ogni operazione eseguita per modificare il contesto grafico ne segue sempre una per ripristinarlo al suo valore precedente in modo da non influenzare con valori e/o attributi errati le rappresentazioni future sotto tutti i punti di vista.

Nell'esempio qui illustrato si può osservare l'istruzione per cambiare il colore e, una volta terminato il riempimento del rettangolo che compone l'i-node, l'istruzione per riportarlo a nero, il quale corrisponde alla colorazione predefinita del contesto grafico all'inizio di ogni applicazione di disegno bidimensionale Java.
L'oggetto ud è un'istanza della classe Utility_disegno, che fondamentalmente contiene metodi utilizzati da ogni pannello e la definizione delle costanti per le dimensioni delle tabelle da rappresentare: infatti sebbene ogni pannello abbia le proprie costanti definite all'inizio della classe, vi sono alcuni elementi comuni che vengono stabiliti da questa classe particolare. Il sistema delle costanti per il dimensionamento dei vari pannelli è stato progettato per semplificare le eventuali operazioni di modifica: qualora infatti si volesse intervenire sul codice per cambiare la posizione dei componenti del disegno, basterebbe cambiare i valori di quelle costanti.
Nella parte di codice relativa alla situazione dopo l'esecuzione della Open, si eseguono delle istruzioni con gli stessi metodi del disegno di base, anche se ovviamente agiscono in modo differente rispetto al precedente metodo, poiché l'immagine risultante è di per se diversa.
Per realizzare invece i metodi relativi ai puntatori e ai collegamenti, sia quelli controllati dalla variabile Disegna_frecce_padre sia quelli controllati dalla variabile Step, si è ricorsi alla classe GeneralPath che permette di definire una traiettoria rettilinea qualunque, secondo le direttive dell'attributo Stroke del contesto grafico corrente.

```
public void puntatore_proc_IO(){
//puntatore chiamante-tabella di sistema
GeneralPath gp= new GeneralPath();

gp.moveTo(DIST_BORDO_SINISTRO+DIST_BORDO_INTERNO_SX+ud.getLARGHEZZA_TAB(),
DIST_BORDO_SUPERIORE_PROC+DIST_BORDOSUP+(ud.getALTEZZA_TAB()/5*3)+(ud.getALTEZZA_TAB()/
10));

gp.lineTo(centroX-
20,DIST_BORDO_SUPERIORE_PROC+DIST_BORDOSUP+(ud.getALTEZZA_TAB()/5*3)+(ud.getALTEZZA_TAB
()/10) );

gp.lineTo(centroX-20, d.height-DIST_BORDO_INFERIORE-
ud.getALTEZZA_TAB()+(ud.getALTEZZA_TAB()/5*3)+(ud.getALTEZZA_TAB()/6));

gp.lineTo(centroX+DIST_CENTRO, d.height-DIST_BORDO_INFERIORE-
ud.getALTEZZA_TAB()+(ud.getALTEZZA_TAB()/5*3)+(ud.getALTEZZA_TAB()/6));

ud.creaFreccia((int)(centroX+DIST_CENTRO-6),(int)( d.height-DIST_BORDO_INFERIORE-
ud.getALTEZZA_TAB()+(ud.getALTEZZA_TAB()/5*3)+(ud.getALTEZZA_TAB()/6)), g2, "destra");

g2.draw(gp);
      }
```

Listato 3.3 Esempio creazione grafica di un puntatore

Il Listato 3.3 presenta i 2 metodi di GeneralPath che permettono la realizzazione della traiettoria sopracitata: *moveTo(),* che posiziona il percorso nel punto indicato come parametro, e *lineTo()*, che garantisce la costruzione di una linea retta partendo dall'ultimo punto della traiettoria corrente e terminando al punto fornito come parametro.
Una volta completato il percorso occorre ordinarne la visualizzazione tramite il metodo *draw().*
Infine il metodo creaFreccia, appartenente alle utility di disegno, consente di creare l'estremità triangolare del puntatore desiderato, combinando i metodi *fill()* e *draw()* con l'orientamento inserito come ultimo parametro.

Completiamo l'analisi del Listato 3.1, sottolineandone la successione di blocchi di codice dovuti al controllo della variabile Step. Ogni blocco deve aggiungere una parte di immagine a quella già rappresentata dagli step precedenti: infatti a causa del metodo *update()*, come è stato già spiegato precedentemente, occorre che ogni step si occupi anche del disegno dei passaggi precedenti.

4 Dettagli sull'implementazione

4.1 Introduzione sul Thread

Il Thread è un concetto che rientra nel meccanismo di programmazione concorrente, ovvero quel tipo di programmazione che garantisce l'esecuzione di attività indipendenti e parallele tra loro. Insieme al thread, l'altra unità base per l'elaborazione in quest'ambito della programmazione è il processo. La differenza con i thread sta nel fatto che un processo ha un ambiente di esecuzione privato (con il proprio codice, il proprio spazio di indirizzamento, il proprio stack, etc...), mentre un thread, considerabile un'unità attiva all'interno di un processo, possiede uno stack riservato ma condivide lo spazio di indirizzamento e le risorse con gli altri thread all'interno del processo che lo ha generato.
Un thread viene alcune volte definito come “processo leggero”, per il semplice fatto che la sua creazione comporta un minore dispendio di risorse rispetto alla creazione di un processo.
Dal punto di vista del linguaggio Java, vengono sfruttati i thread nel campo della programmazione concorrente.

Un thread in Java può essere implementato in 2 modalità:

1) utilizzando una classe che estende la classe Thread;
2) utilizzando una classe che implementi l'interfaccia Runnable.

Entrambe le scelte generano un metodo *run()* che contiene il corpo del thread, ovvero le istruzioni che esso deve eseguire quando viene chiamato il metodo *start()* sull'istanza della classe utilizzata.
Il metodo *start()*, infatti, indica al thread di cominciare l'esecuzione delle sue istruzioni in modo del tutto parallelo all'esecuzione della parte di codice corrente da parte dell'applicazione. Il thread può essere fermato con il metodo *stop(),* oppure esso termina autonomamente una volta conclusi i compiti all'interno del metodo *run()*.

4.2 Interazione tra Thread e GUI

Per far si che l'interfaccia grafica risulti sempre responsiva, è necessario considerare l'interazione tra la stessa e i thread. Nell'utilizzo dei componenti Swing il thread di riferimento è chiamato EDT (Event Dispatch Thread): compito di questo thread, fornito dalla libreria Swing, è quello di rappresentare e mostrare a video l'interfaccia e di gestire gli eventi generati dalla GUI in seguito a determinate azioni da parte dell'utente. L'EDT è avviato automaticamente dalla Java VM (Virtual Machine) nel momento in cui viene richiamato un metodo *paint()* (ovvero quando un componente deve essere disegnato) e, durante l'esecuzione dell'applicazione, agisce sulla GUI gestendo una coda degli eventi generati dalla GUI stessa: prelevando infatti un evento da questa coda, l'EDT notifica al Listener corrispondente il tipo di evento ed esegue il codice per la gestione dell'evento prelevato; quindi tutto il codice presente nei metodi actionPerformed() o mouseClicked(), giusto per citare 2 esempi utilizzati in questa applicazione, è eseguito nell'EDT.
Per evitare dunque di congelare la GUI e renderla non responsiva è necessario che i codici per la gestione degli eventi non siano bloccanti o troppo onerosi per l'EDT.
Finchè c'è almeno un evento in coda da gestire o finchè un componente, di solito un container, è visibile a video, l'EDT viene mantenuto attivo.
Sull'EDT quindi viene eseguito il codice per la gestione degli eventi e il codice che invoca metodi relativi a componenti Swing. Questo è necessario perchè la maggior parte dei metodi Swing non è “Thread Safe”, ovvero invocando i metodi da più thread si può incorrere nel rischio di deadlock o di inconsistenza a livello della memoria. Ci sono stati alcuni tentativi da parte dei programmatori per scrivere delle interfacce grafiche cosiddette multithread, ma per i motivi sopracitati si è deciso di utilizzare un modello ad unico thread che realizzasse un gestore della coda degli eventi.

Esistono comunque alcune eccezioni a questo modello: vi sono infatti dei metodi Swing che sono “Thread Safe” e che quindi posso essere invocati da qualuque thread senza problemi. L'indicazione “Thread Safe” è presente nelle API Java dei metodi. Ad esempio nel metodo *append()* del componente JTextArea si può osservare la seguente frase : *“This method is thread safe, although most Swing methods are not “.*
È quindi buona norma sottostare a questa specie di regola, ovvero far sì che ogni operazione che consiste nella visualizzazione, modica o aggiornamento di un componente Swing venga eseguita nell'EDT.
Ci si potrebbe chiedere dunque perchè sia stata effettuata la scelta di non rendere “Thread Safe” la libreria Swing. In effetti ci sono alcune ragioni che hanno portato a questa scelta:

1) realizzare componenti “Thread Safe” avrebbe introdotto costi aggiuntivi per la sincronizzazione e la gestione dei lock sulle risorse, riducendo la velocità e la reattività della libreria, mentre una GUI deve essere rapida e responsiva;
2) dal punto di vista del programmatore, la complessità delle applicazioni da sviluppare sarebbe aumentata notevolmente nel caso di componenti “Thread Safe”, soprattutto per via di tutte le problematiche legate all'interazione tra i thread; al programmatore sarebbe stata richiesta una perfetta conoscenza del multithreading per scrivere applicazioni Swing.

Quindi limitando l'accesso ad un solo thread, la libreria Swing offre prestazioni elevate e non richiede che il programmatore abbia conoscenze molto avanzate di multithreading, garantendo una gestione sequenziale degli eventi.

Esiste inoltre la possibilità di interagire direttamente con l'EDT: tramite la classe SwingUtilities, infatti, si può forzare l'EDT ad eseguire codice dell'applicazione a discrezione del programmatore. I metodi messi a disposizione da questa classe sono *InvokeLater()* e *InvokeAndWait()* ed entrambi richiedono in input un oggetto Runnable. Il corpo del metodo run() deve contenere il codice che si vuole far eseguire all'EDT (tipicamente relativo alla modifica, creazione o aggiornamento di componenti Swing). Il metodo *InvokeLater()* consente di richiedere l'esecuzione del codice per poi continuare immediatamente la sue operazioni, senza attendere che l'EDT abbia terminato l'esecuzione, mentre il metodo *InvokeAndWait()* attende che il codice venga eseguito dall'EDT, o meglio il thread che ha invocato il metodo resta in attesa dell'esecuzione nell'EDT. In entrambi i casi il riferimento all'istanza dell'oggetto Runnable viene inserito nella coda degli eventi e quando questo riferimento viene prelevato l'EDT ne esegue il metodo *run()*.

Esempio.

```
//lancio del programma forzando la creazione della GUI nell'EDT
SwingUtilities.invokeLater(new Runnable() {
public void run() {

Frame_generale f= new Frame_generale();

   }
});
```

Come si può notare dall'esempio, nell'applicazione è stata utilizzata la possibilità offerta dalle SwingUtilities per forzare la costruzione iniziale della GUI nell'EDT: infatti il thread che lancia il programma, ovvero quello che richiama il metodo *main()*, dovrebbe occuparsi anche della creazione della GUI e, per il discorso fatto precedentemente, è preferibile far eseguire tutto il codice legato ai componenti Swing all'EDT.

4.3 Esempio di utilizzo di un Thread nell'applicazione

Nell'applicazione è stato utilizzato un thread per il supporto all'opzione sulle informazioni aggiuntive (vedere paragrafo 5.2). Si è deciso di ricorrere ad un thread aggiuntivo per evitare il blocco della GUI: infatti come accennato nel paragrafo precedente, occorre evitare di far compiere all'EDT delle operazioni bloccanti, potenzialmente tali o semplicemente troppo onerose.
Le operazioni che il thread deve compiere sono relative alla visualizzazione su un'apposita finestra delle informazioni prelevate da un file. Il file scelto varia a seconda della primitiva selezionata dall'utente.
Passiamo ad analizzare nel dettaglio i vari passi che portano al risultato finale (osservabile tramite la Figura 4.1, equivalente a quella mostrata nel prossimo capitolo).

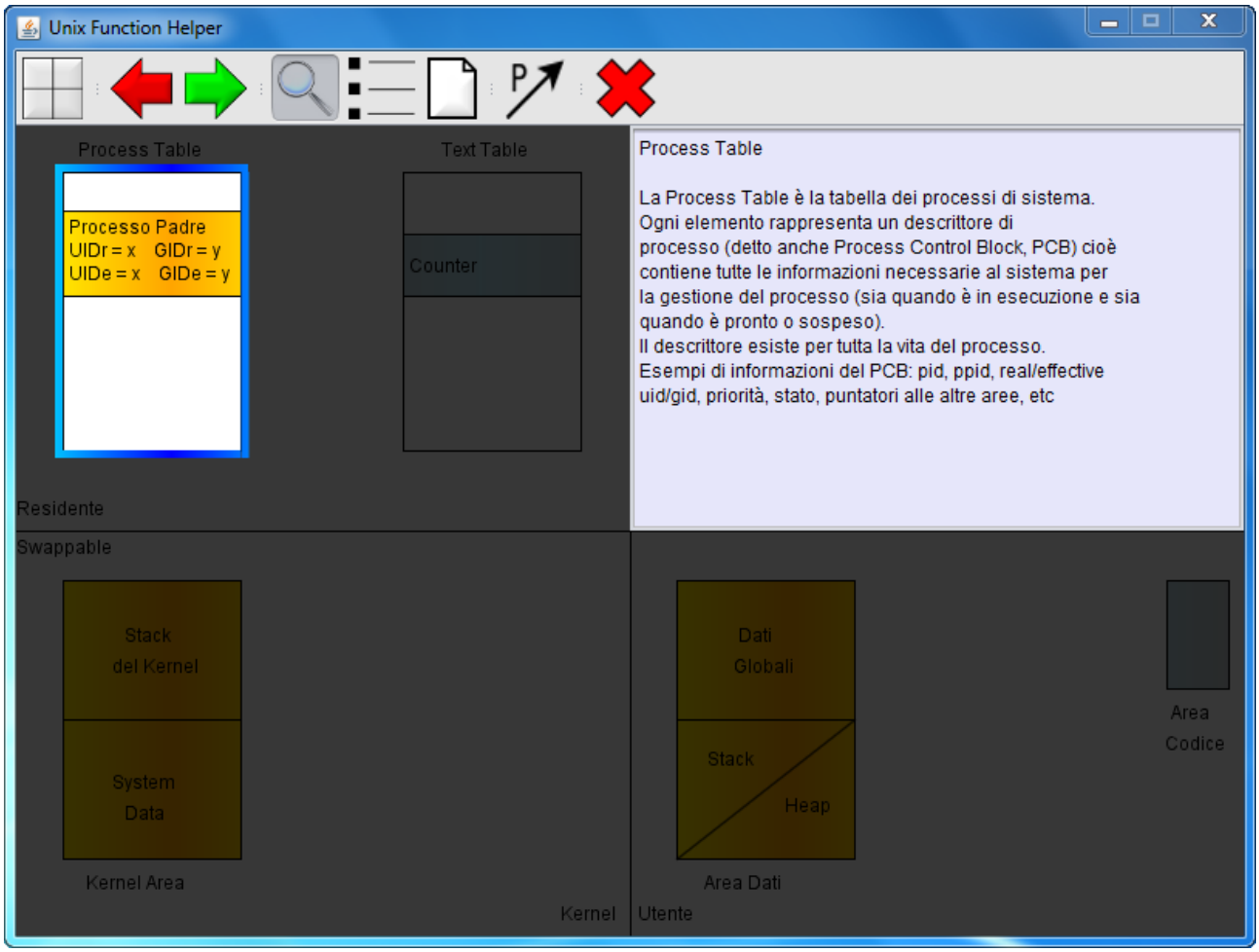

Figura 4.1 Risultato delle operazioni compiute dal thread

Il thread è organizzato in un ciclo infinito esterno ed uno interno. Nel ciclo interno esso esegue una sleep e al momento del risveglio controlla se è arrivata qualche indicazione da parte dell'istanza parent_frame della classe Frame_generale: come già spiegato nel precedente capitolo l'istanza parent_frame è il centro di tutte le variabili di stato del programma e, tra queste, è presente una variabile chiamata azione_thread che, quando le viene assegnato il valore 1, è in grado di sbloccare il thread dal ciclo infinito interno (Listato 4.1), permettendogli di continuare l'esecuzione.

```
for(;;){
[…]
Thread.sleep(100);
if(i==null){
i= new Informazioni(parent_frame);}
```

```
if(parent_frame.getAzione_thread()==1){
[…]
break;
}
}
```

Listato 4.1 Estratto del ciclo infinito interno

L'istanza i è la finestra che permette di mostrare a video le informazioni, mentre le istruzioni non riportate riguardano solamente dei passaggi per ricavare le giuste coordinate in modo da poter correttamente posizionare la finestra di visualizzazione. La variabile azione_thread viene posta a 1 nel momento in cui l'utente richiede le informazioni su un particolare elemento del disegno con un click del mouse, quindi in ogni pannello è presente questo tipo di settaggio.
Una volta uscito dal ciclo interno, il thread entra in un while (il cui corpo è il resto del thread stesso) e controlla, sempre attraverso parent_frame, altre 2 variabili:

1) quella relativa al nome della primitiva corrente (Primitiva), per caricarne il corrispondente file;
2) quella relativa all'elemento scelto dall'utente (Evidenzia), per caricare dal file selezionato la giusta voce.

Il metodo che agisce sulla variabile Evidenzia è il metodo *mouseClicked()* dell'interfaccia MouseListener implementata in ogni pannello. Questo metodo infatti a seconda della coppia di coordinate che ritorna l'evento generato dal mouse, è in grado di ricavare se la coppia si trova all'interno di uno dei componenti del disegno.

```
//se viene selezionato un componente del disegno si predisponde il caricamento del file
if(!(parent_frame.getEvidenzia().equals(""))){
//caricamento del file primitiva.txt e lettura con lo scanner
url=getClass().getResource("/File/"+parent_frame.getPrimitiva()+".txt");
}


try {
//apertura di una connessione al file tramite l'url
scanner = new Scanner(url.openStream(),"UTF-8");
} catch (IOException e) {
e.printStackTrace();
}
```

Listato 4.2 Predisposizione dello Scanner

Nel Listato 4.2 viene mostrata la preparazione dell'operazione di lettura: oltre al meccanismo del *getResource()* analizzato precedentemente, dato che anche i File sono risorse contenute nel Jar, viene utilizzata un'istanza della classe Scanner. Lo scanner richiede in ingresso un Input Stream per iniziare a leggere, ma nel listato si nota un secondo parametro, quello relativo al tipo di encoding dei caratteri del file: è vero che Java utilizza lo standard di codifica Unicode, ma le rappresentazioni in memoria sono molto differenti e per questo si parla di diversi tipi di encoding. Per poter rendere del tutto portabile l'applicazione non si poteva lasciare la rappresentazione dei caratteri ad ogni singolo sistema operativo secondo il proprio encoding, perchè vi sono alcune codifiche che mappano solo una parte dei caratteri Unicode, mentre i rimanenti vengono mappati nel carattere ? (è il caso della codifica CP1252 tipica di alcuni sistemi Windows) e quindi nel passaggio tra architetture differenti ci potevano essere errori di visualizzazione.
Per questo motivo è stato deciso di scrivere i file nel sistema di encoding UTF-8 (uno dei più utilizzati) e di imporre allo scanner di trattarli in lettura secondo quella codifica, in modo tale che

sotto qualunque sistema operativo possano comparire a video le stesse lettere.

Una volta preparato lo scanner, il thread inizia la lettura cercando la tabella di interesse secondo la variabile Evidenzia e esegue una scissione distinguendo tra vari casi:

1) controlla se la primitiva in questione non ha effetto sulla tabella selezionata: in questo caso andrà stampata a video la normale descrizione della tabella in entrambe le situazioni della primitiva;
2) se la primitiva ha effetto sulla tabella, il controllo chiaramente avviene sulle variabili del parent_frame per capire in che situazione si trova l'applicazione al momento (se prima o dopo l'azione della primitiva);
3) viene eseguito anche un ulteriore controllo sul fatto che sia abilitata l'opzione per la visualizzazione graduale degli effetti della primitiva (vedi Figura 5.6 e 5.7 nel paragrafo 5.2), e si valuta se mostrare gli effetti della primitiva o meno a seconda del numero di step che l'utente ha già visualizzato.

Stabilito la tipologia di informazione che deve essere prelevata, il thread scorre il file tramite lo scanner finchè non trova un match con la stringa corrispondente alla variabile Evidenzia di parent_frame, ovvero la variabile che contiene il nome della tabella selezionata dall'utente.
La finestra a cui si appoggia il thread per visualizzare le informazioni presenta una JTextArea e quindi ogni nuova riga letta dallo scanner viene semplicemente aggiunta in append a quelle presenti. Nel file sono present le stringhe “end_tab” e “Effetto della Primitiva”per indicare allo scanner di terminare la propria operazione di lettura sequenziale.

```
if(parent_frame.getDisegna()==2 && parent_frame.getSc().getStep()==0){
while(!(line.equals("Effetto della primitiva:"))){

line=scanner.nextLine();
}
//stampiamo il titolo della tabella e lasciamo un po' di spazio
i.getArea().append(parent_frame.getEvidenzia());
i.getArea().append("\n");
i.getArea().append("\n");



while(!(line.equals("end_tab"))){

//la linea è corretta quindi viene scritta sull'area di testo
i.getArea().append(line);
//andiamo a capo per rispettare la formattazione del file di testo
i.getArea().append("\n");
//passiamo alla prossima riga
line=scanner.nextLine();
}
```

Listato 4.3 Lettura e stampa nel caso in cui la variabile Disegna sia uguale a 2

Il Listato 4.3 è un esempio di lettura e stampa a video delle informazioni prelevate dal file, sfruttando lo scanner e l'area di testo: viene fatto un ciclo che salta le righe prima dell'effetto della primitiva (dal momento che siamo nella rappresentazione della situazione dopo l'azione della primitiva stessa) e di seguito un ciclo di diversi append finchè lo scanner non rileva la stringa “end_tab”.
Terminata la scrittura, si entra nell'ultimo blocco di codice dove viene chiuso lo scanner e la finestra viene resa visibile e posizionata in base ai calcoli fatti nel ciclo infinito interno visto precedentemente.

```
else if(trovata==1){
scanner.close();
scanner=null;
trovata=0;
parent_frame.setAzione_thread(0);
i.validate();
//condizioni per posizionare l'area di testo in base al componente selezionato
[...]
i.setVisible(true);
break;
}
```

Listato 4.4 Parte finale del while contenuto nel ciclo infinito esterno

Con il listato 4.4 si conclude l'analisi del thread: la variabile trovata serve per evitare di continuare a leggere dopo aver già trovato quello che serve all'interno del file; il parametro che sveglia il thread dal ciclo infinito interno viene rimesso a zero in modo da far ritornare il thread stesso in quel ciclo e, dopo il corretto posizionamento della finestra delle informazioni, il while termina e il thread ritorna nel ciclo infinito interno, in attesa di una nuove informazioni da visualizzare

5. Descrizione dell'applicazione

5.1 Introduzione

L'applicazione è stata realizzata con i componenti delle librerie Swing e AWT di Java. È una GUI (Graphical User Interface) costituita principalmente da un JFrame ed una JToolbar disposta secondo il BorderLayout Manager nella parte superiore del container. Il JFrame ha una dimensione di 800x600 pixel.
Lo spazio rimanente all'interno del frame (visibile nella Figura 5.1), vuoto al primo avvio del programma, viene utilizzato di volta in volta dai vari pannelli che mostrano l'evolversi dell'azione delle 4 primitive.
Ogni azione che l'utente può compiere è collegata ad un bottone nella JToolbar, e a seconda della primitiva selezionata compaiono bottoni differenti. Alcune azioni però sono comuni a tutte le primitive quindi i corrispondenti bottoni rimangono immutati. Ad ogni bottone è associato, nella parte di codice, un metodo *repaint()* che, come visto precedentemente, permette di ridisegnare il pannello vuoto e consente all'utente di osservare il risultato dell'opzione selezionata.
Su ogni bottone, inoltre, è presente un ToolTip: se l'utente non preme un determinato bottone ma semplicemente mantiene il cursore del mouse fisso su di esso, si assiste alla comparsa di un piccolo riquadro che indica l'operazione che è possibile attivare con la selezione del bottone stesso.

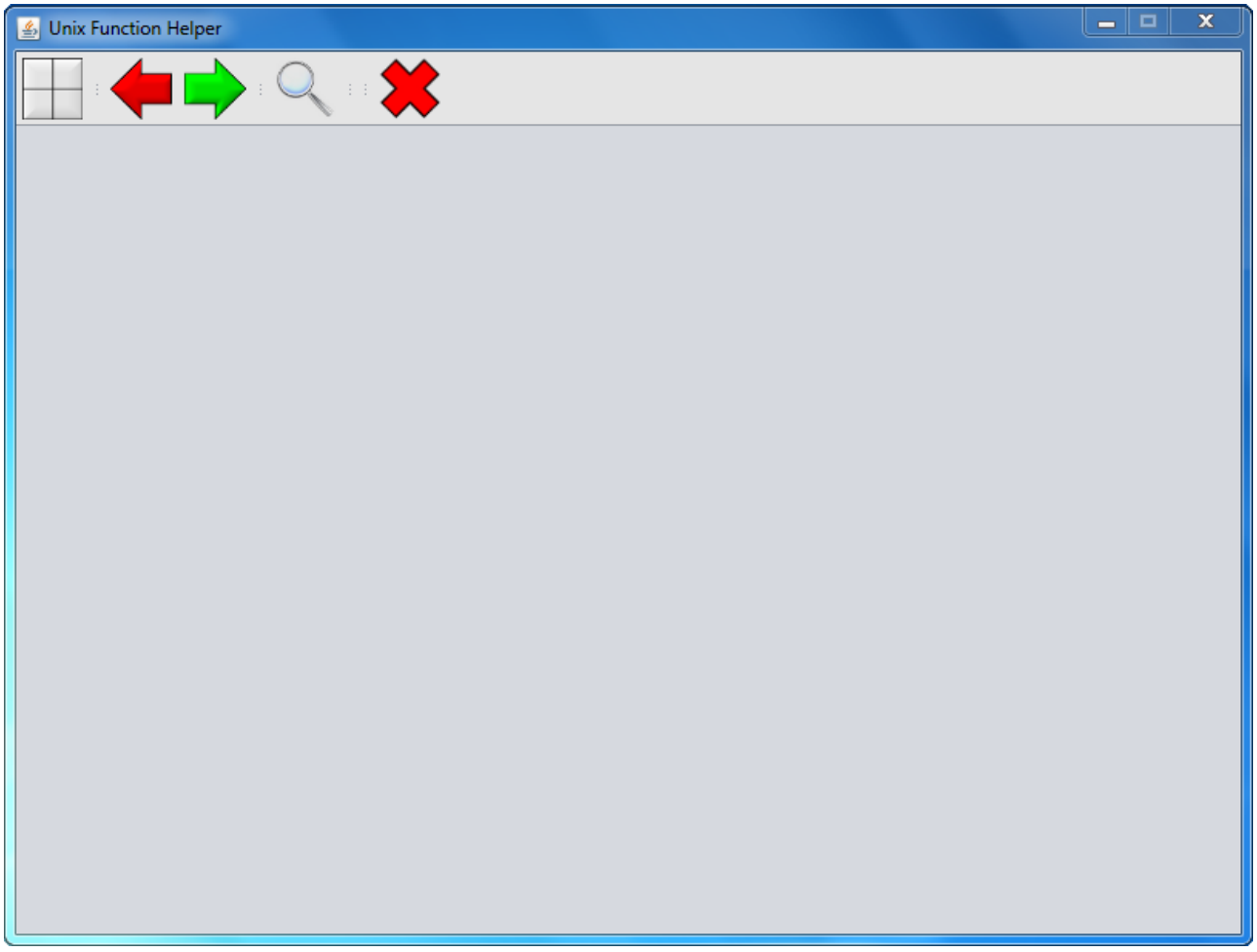

Figura 5.1 GUI all'avvio dell'applicazione.

Nel paragrafo successivo verranno prese in esame tutte le possibili opzioni che l'utente può utilizzare, ricorrendo alla primitiva Fork come esempio visivo.

5.2 Spiegazione della GUI

La prima opzione da considerare è certamente quella che l'utente deve adoperare per cominciare a riempire il pannello vuoto della GUI che gli si presenta all'avvio dell'applicazione: l'opzione di selezione della primitiva. Il relativo bottone è il primo a sinistra all'interno della JToolbar (quello a forma di quadrato suddiviso in 4) e porta alla comparsa di una maschera intermedia (come mostrato in Figura 5.2), che presenta la suddetta selezione sotto forma di 4 bottoni disposti in un rettangolo. Non è possibile accedere alle altre funzionalità del pannello se non viene prima selezionata una primitiva.

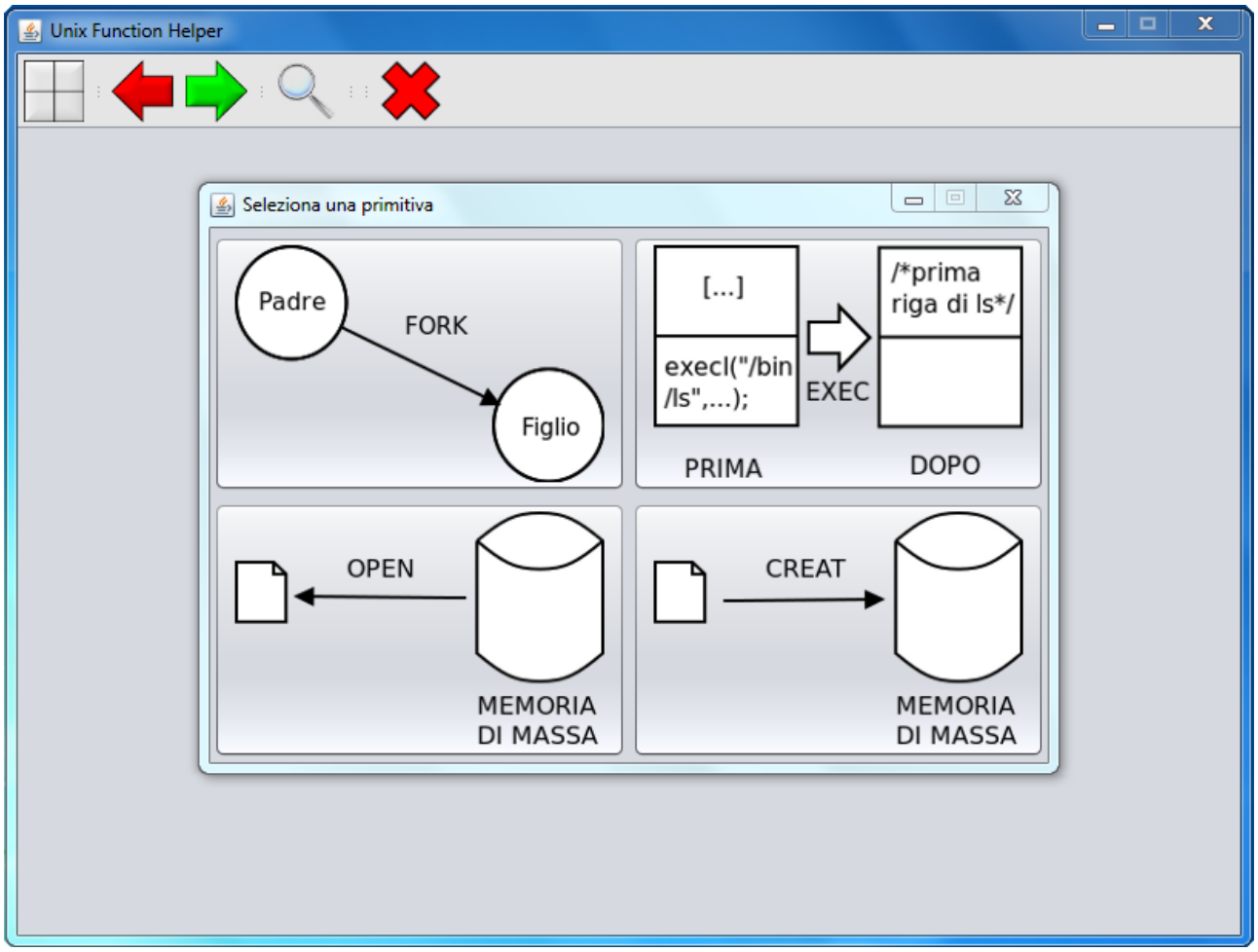

Figura 5.2 Maschera per la selezione della primitiva

5.2.1 La primitiva Fork

La seconda opzione che si presenta scorrendo la JToolbar verso destra è quella che permette di visualizzare la situazione antecedente all'azione della primitiva selezionata. È attivabile cliccando il bottone che ha come icona la freccia rossa che punta verso sinistra. È la situazione di partenza per ogni primitiva, ed è infatti quella che viene normalmente mostrata nel momento in cui l'utente clicca uno dei 4 bottoni della maschera intermedia, senza dover necessariamente premere il bottone sopracitato.

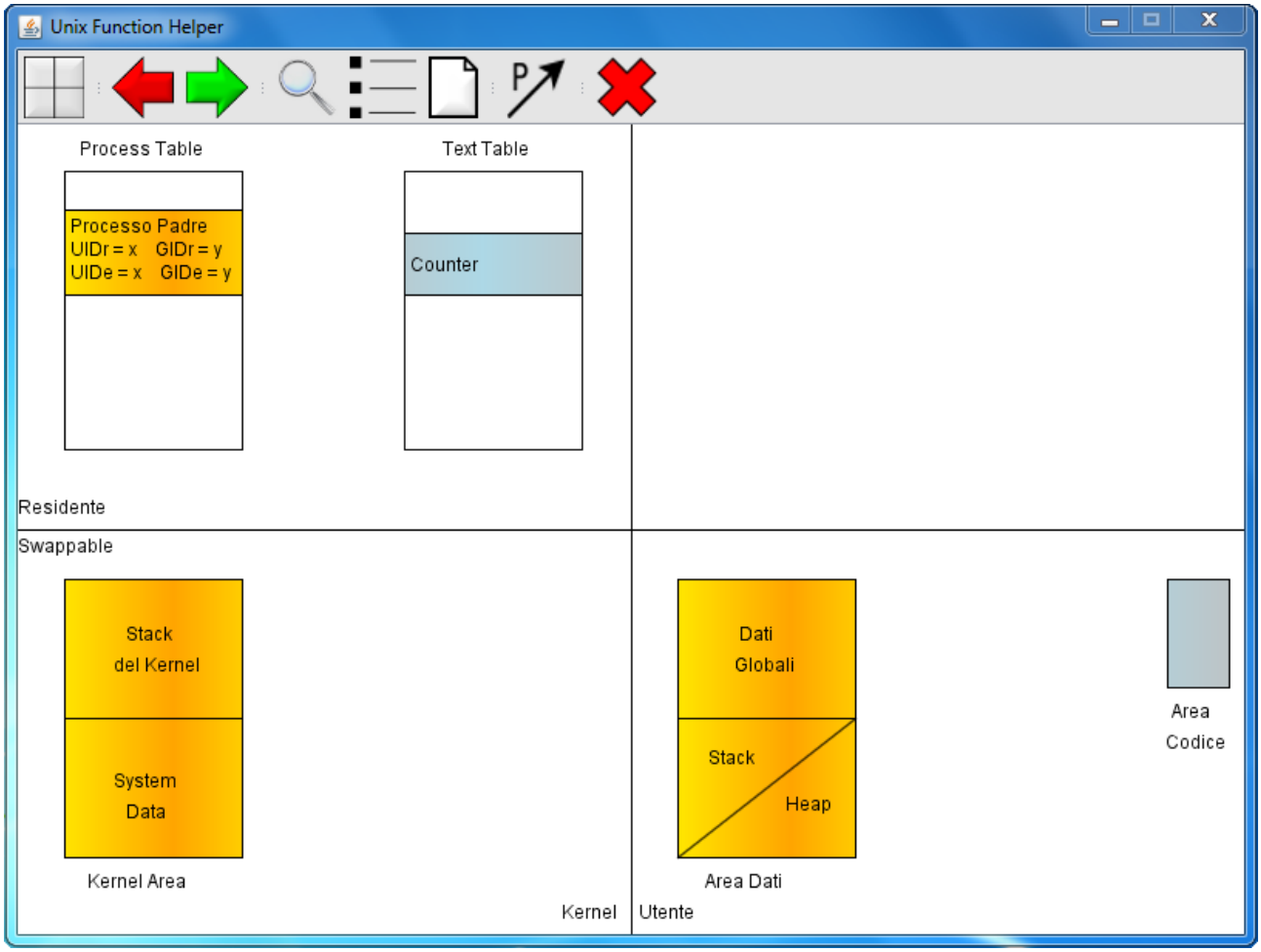

Figura 5.3 Situazione di partenza prima dell'azione della primitiva Fork

Poichè l'immagine di Figura 5.3 è relativa alla primitiva Fork, sono rappresentati i componenti su cui essa interviene, quindi il pannello è stato suddiviso in 4 parti per evidenziare le 4 zone di memoria relative al processo padre: kernel residente, kernel swappable, utente residente ed utente swappable.
Sono state posizionate alcune tabelle e l'area del codice, a cui fa riferimento il processo padre, nelle corrispondenti aree.

È sempre possibile tornare a questo disegno semplicemente cliccandone il bottone relativo, qualunque sia l'opzione corrente scelta dall'utente, ad eccezione del caso in cui questo bottone non sia disabilitato dall'opzione stessa.

La prossima opzione da analizzare è quella che permette di mostrare a video gli effetti dell'azione della primitiva: il bottone relativo a questa opzione è quello la cui icona è una freccia verde che punta verso destra. La Figura 5.4 è sempre un'immagine legata alla primitiva Fork: in questo caso infatti si nota la comparsa del processo figlio con le corrispondenti tabelle (indicate utilizzando la stessa colorazione del processo figlio) nelle rispettive locazioni.

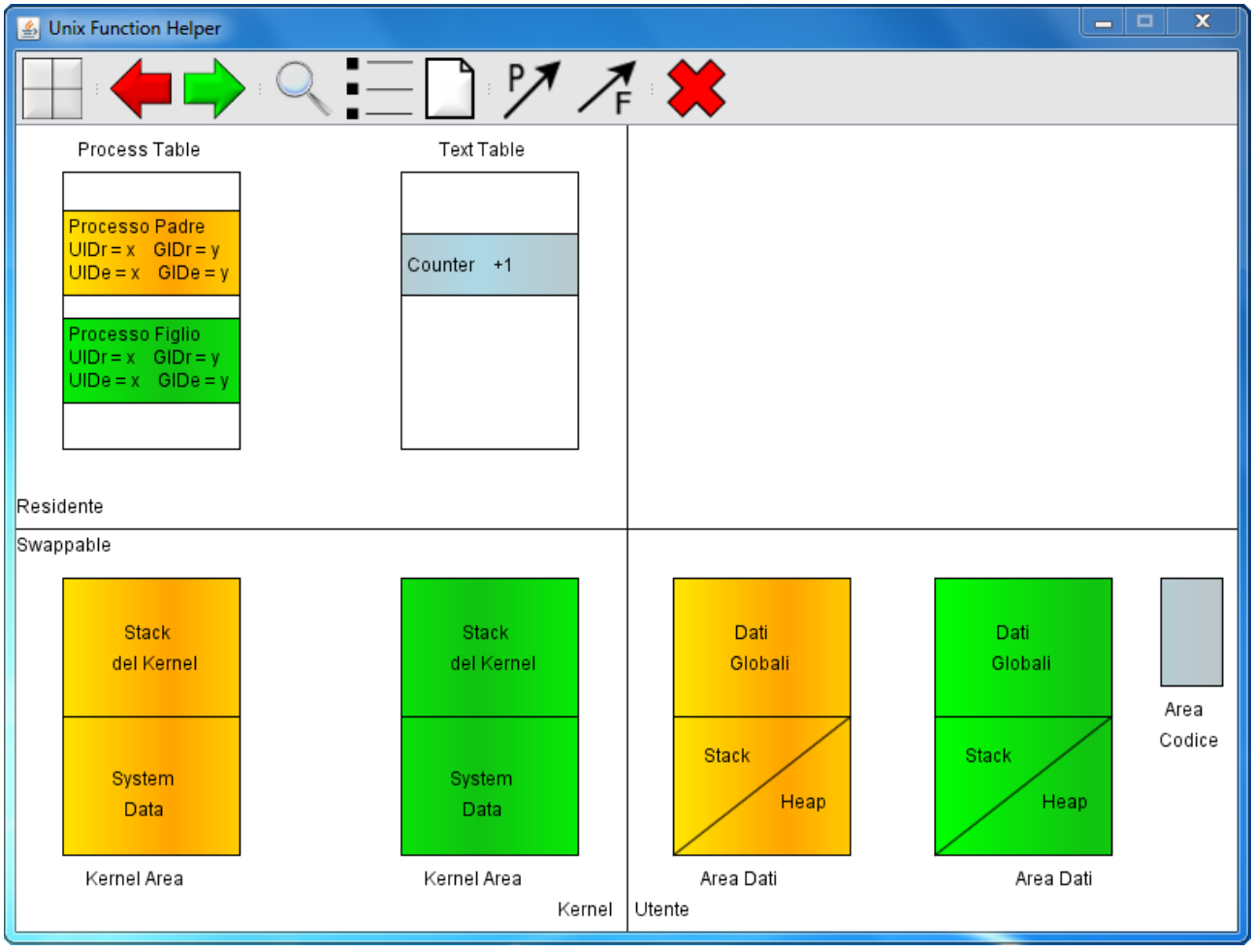

Figura 5.4 Situazione visualizzata dopo l'azione della primitiva Fork

Questa è una delle opzioni che permettono di far comparire o scomparire possibilità aggiuntive: nel caso della Fork, infatti, è comparso un bottone che per icona presenta una freccia nera sopra alla lettera F (Figlio) di fianco alla freccia nera sotto la lettera P (Padre). Queste frecce mostrano i puntatori del processo figlio/padre verso tutte le aree di memoria a esso legate. Analogamente premendo il bottone dell'opzione precedente, ovvero quella relativa alla situazione antecedente l'azione della primitiva, questa possibilità di visualizzazione aggiuntiva scompare, poiché non c'è nessun processo figlio da rappresentare.

La prossima opzione, invece, è una delle principali caratteristiche dell'applicazione offerta all'utente. Il bottone che come icona presenta la lente di ingrandimento garantisce la comparsa di informazioni aggiuntive su un qualunque componente del disegno corrente. Questo bottone è di tipo Toggle, ovvero rimane selezionato quando viene cliccato, mostrando così che l'opzione è stata attivata. Dal momento in cui viene premuto è possibile cliccare un elemento nel disegno per ottenere le informazioni; la schermata che si presenta è simile a quella mostrata in figura 5.5: in questo caso è stata seleziona la Process Table nella situazione antecedente l'azione della Fork, e il risultato è un effetto che mette in risalto questa tabella oscurando il resto del disegno. In seguito viene resa visibile la descrizione della Process Table tramite un componente molto simile al JFrame, che non contiene però tutte le decorazioni del Frame stesso, ma solo il pannello centrale.
La particolarità della visualizzazione delle descrizioni riguarda il fatto che questa finestra si posiziona sul pannello a seconda della locazione della tabella a cui si riferisce: questo avviene per evitare di coprire involontariamente la tabella selezionata.

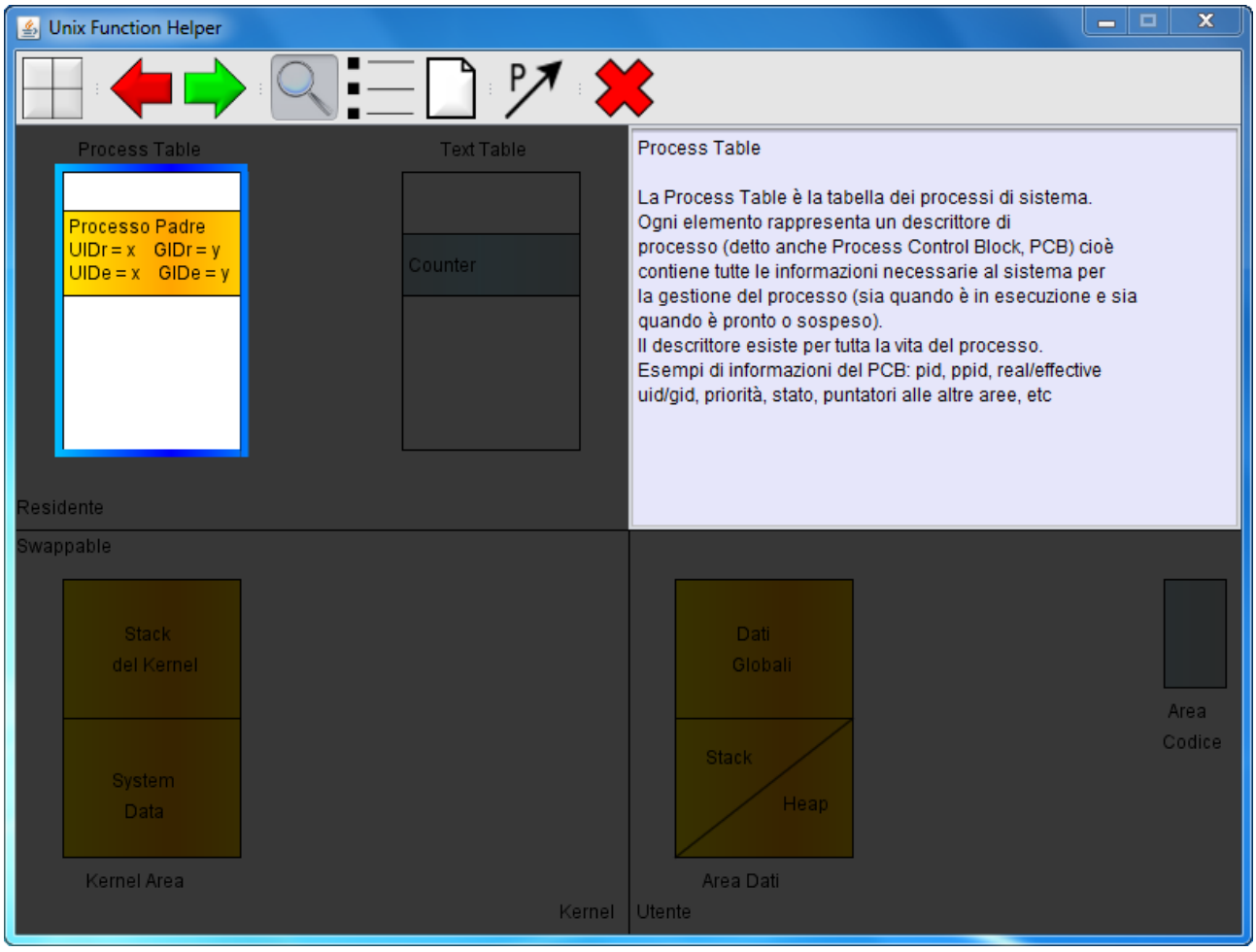

Figura 5.5 Opzione per le informazioni aggiuntive sugli elementi dell'immagine

Cliccando su un'area vuota del pannello l'effetto dell'evidenziatore sella tabella scompare e la descrizione viene nascosta. È possibile continuare a scegliere altre tabelle o altri elementi del disegno se si vogliono conoscere altri dettagli. Le informazioni visualizzate cambiano a seconda della situazione in cui ci si trova: se la situazione è precedente all'azione della primitiva allora verrà mostrata una descrizione dell'elemento, altrimenti verrà illustrato l'effetto che ha la primitiva sull'elemento stesso. Cliccando nuovamente il bottone si disattiva questa opzione.

Il bottone successivo, anch'esso Toggle, è relativo all'opzione per la visualizzazione degli effetti della primitiva in passaggi successivi: essa permette tramite una finestra guida, che compare alla sinistra del frame, di scorrere le varie azioni che la primitiva compie a sui vari elementi del disegno. È possibile percorrere anche a ritroso i vari step interagendo con i 2 bottoni presenti nella finestra di supporto. La finestra di supporto (figura 5.7) scompare solamente se viene cliccato il bottone dell'opzione. A seconda della primitiva gli step successivi mostrano l'intero disegno che si avrebbe cliccando l'opzione sulla situazione dopo l'azione della primitiva, oppure nascondono alcuni elementi per maggiore chiarezza: è il caso dell'esempio con la primitiva Fork in Figura 5.6, dove non vengono mostrati i puntatori del processo padre.

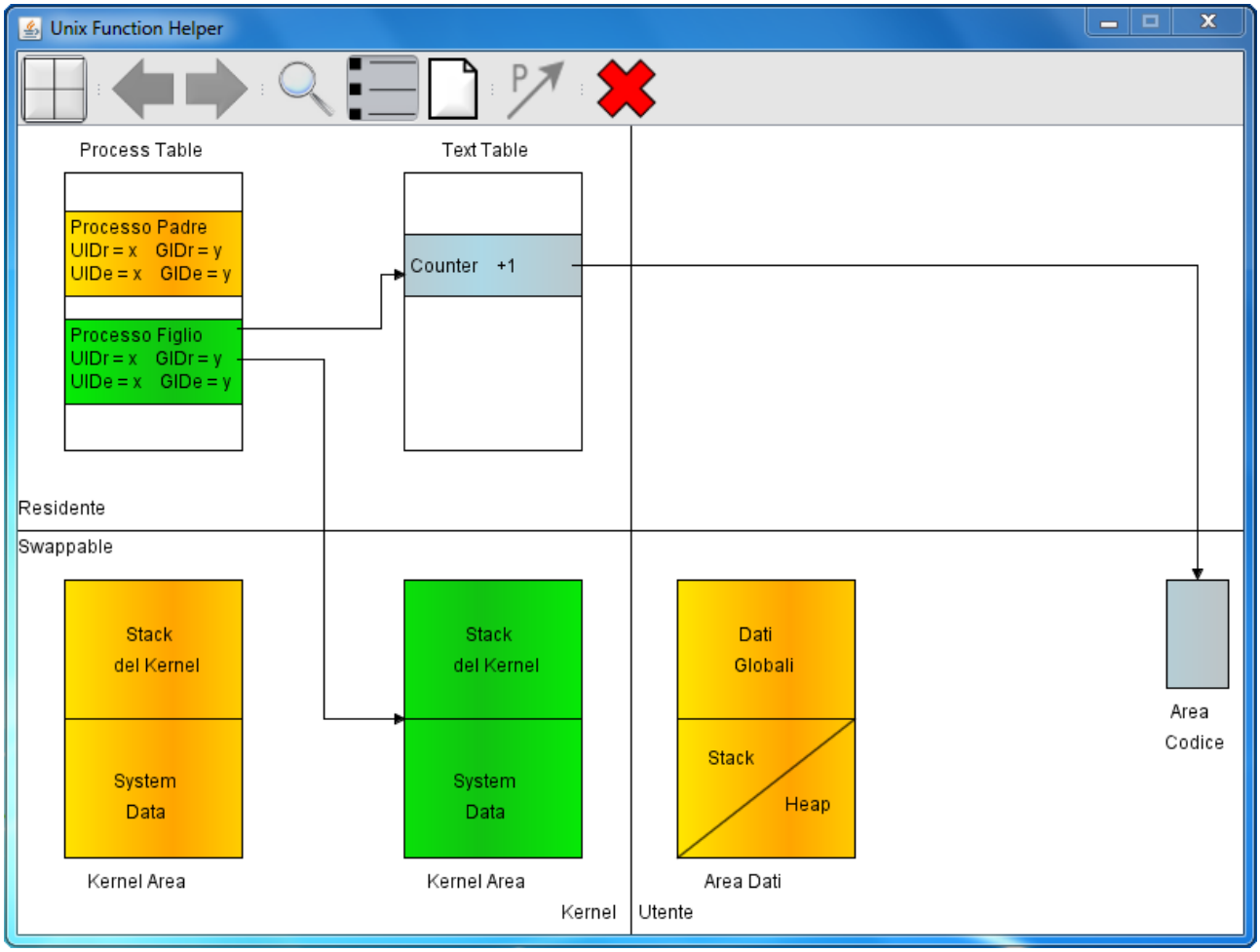

Figura 5.6 Opzione per la visualizzazione graduale dell'azione della primitiva Fork

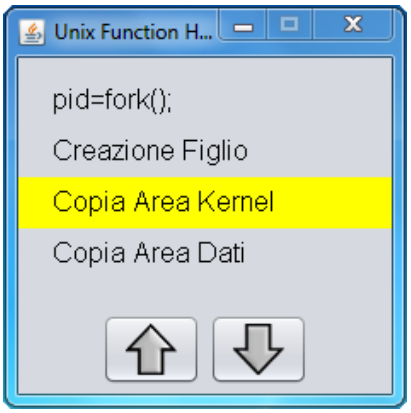

Figura 5.7 Finestra di supporto

Questa opzione disabilita la possibilità di utilizzare i bottoni relativi ai puntatori e quelli relativi al cambio di situazione dal momento che ogni elemento utile viene mostrato nei vari step.
Infine se l'utente decide di sfruttare l'opzione per le informazioni aggiuntive, descritta precedentemente, visualizzerà gli effetti delle primitive solo nel caso in cui l'elemento selezionato abbia già subito l'azione dello step ad esso collegato, altrimenti verrà mostrata la normale descrizione del componente.

La penultima opzione, relativa sempre ad un bottone Toggle, riguarda la possibilità di visualizzare una certa operazione riguardo ad un file. Quest'operazione assume un significato diverso a seconda della primitiva che si sta esaminando: nell'esempio della Fork, essa mostra l'esecuzione che si verifica se, al momento della creazione del processo figlio, è presente un file aperto nel processo padre. Il disegno viene completamente cambiato perchè l'analisi si sposta su altre tabelle di sistema e dei processi, come mostrato in Figura 5.8.

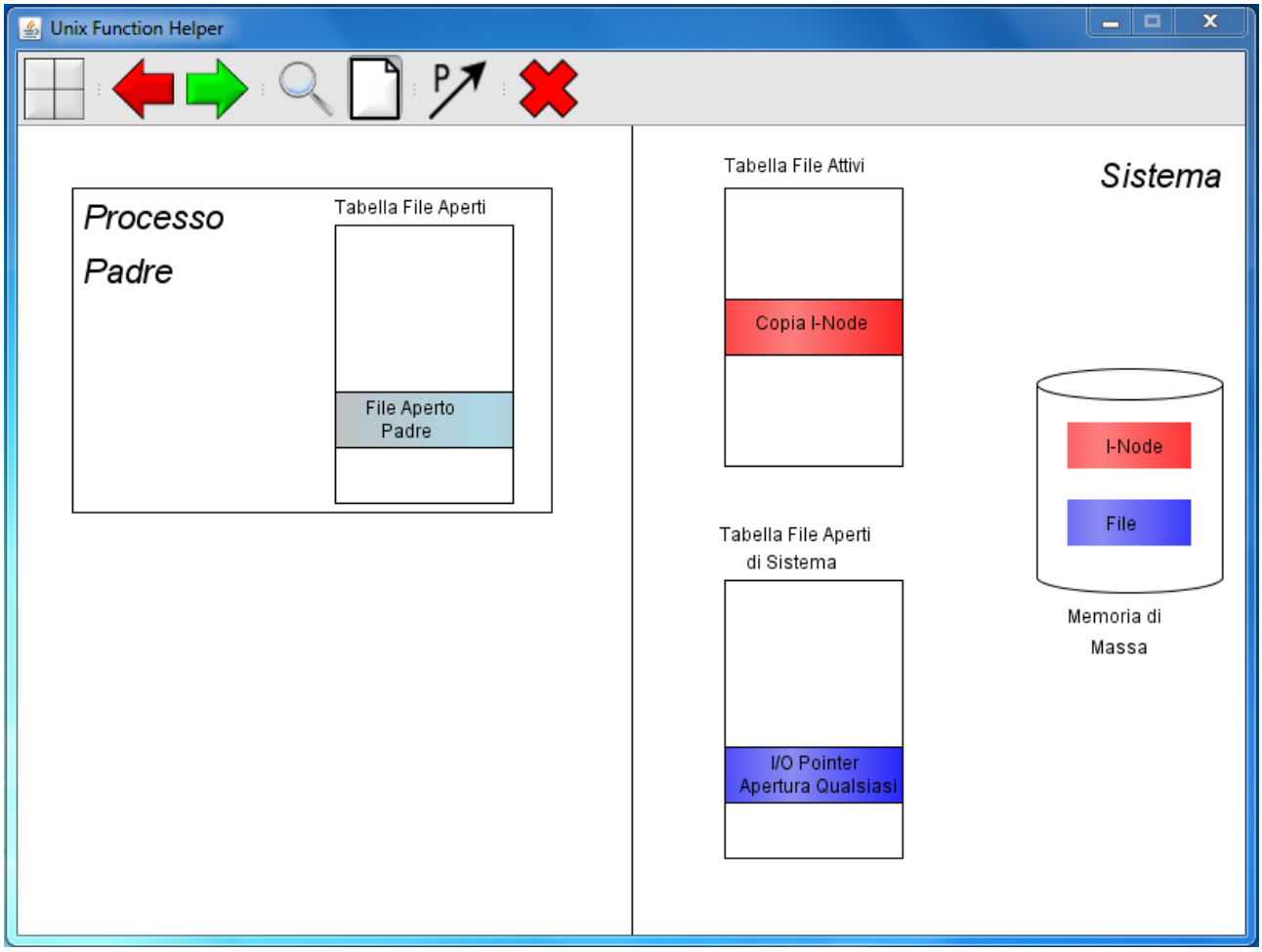

Figura 5.8 Opzione per inserire un file aperto nel processo padre

Per quanto riguarda la primitiva Creat, l'opzione permette di decidere se il file oggetto della primitiva esiste o meno.
Questa opzione, nella primitiva Fork, disabilita l'opzione per la visualizzazione graduale degli effetti perchè la situazione cambia in un unico passaggio dopo l'azione della primitiva.

Infine abbiamo l'opzione per la visualizzazione dei puntatori. È possibile avere un secondo bottone relativo ai puntatori nel caso delle primitive Fork ed Exec per quanto riguarda il processo figlio. In questi casi l'utente può abilitare il disegno sia dei puntatori del padre sia di quelli del figlio, oppure può abilitare solo quelli del padre/figlio, dal momento che entrambi i bottoni sono di tipo Toggle.
L'esempio della primitiva Fork prima della creazione del figlio (senza quindi il secondo bottone per i puntatori) è mostrato in Figura 5.9

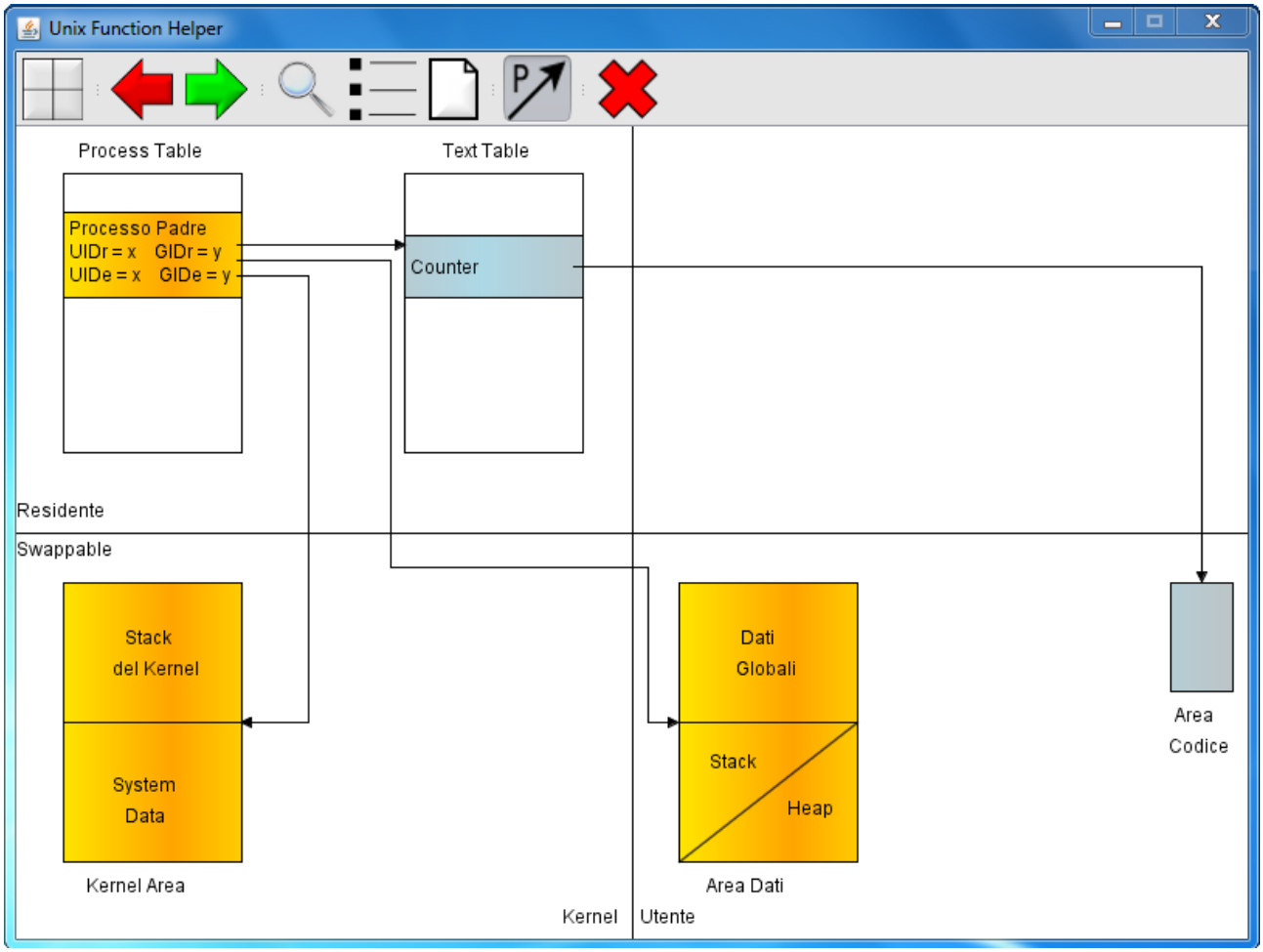

Figura 5.9 Opzione per mostrare i puntatori del processo padre verso i componenti dell'immagine

L'ultimo bottone presente nella Toolbar chiude l'applicazione, indipendentemente dal disegno e dalle modalità di visualizzazione impostate.

Completiamo ora la spiegazione della GUI con una veloce panoramica delle altre 3 primitive mostrando le differenze nei disegni con la Fork.

5.2.2 La primitiva Exec

Per quanto riguarda la primitiva Exec abbiamo 2 possibili modalità di visualizzazione, dipendenti dall'utilizzo o meno del bottone F, di tipo Toggle, a sinistra del bottone per i puntatori del processo padre. La F sta per Fork, ed infatti quell'opzione garantisce la combinazione della primitiva Exec con la Fork, mostrando che è il processo figlio quello che si occupa dell'esecuzione del programma passato come parametro all'Exec stessa. Dal punto di vista dell'immagine vera e propria, quella che si presenta al momento della selezione della primitiva, senza far uso del bottone Fork, è sostanzialmente identica alla Figura 5.3 ad eccezione della scritta processo chiamante invece di processo padre. La Figura 5.10, invece, mostra l'effetto della primitiva Exec se il punto di partenza è la situazione senza il bottone Fork.

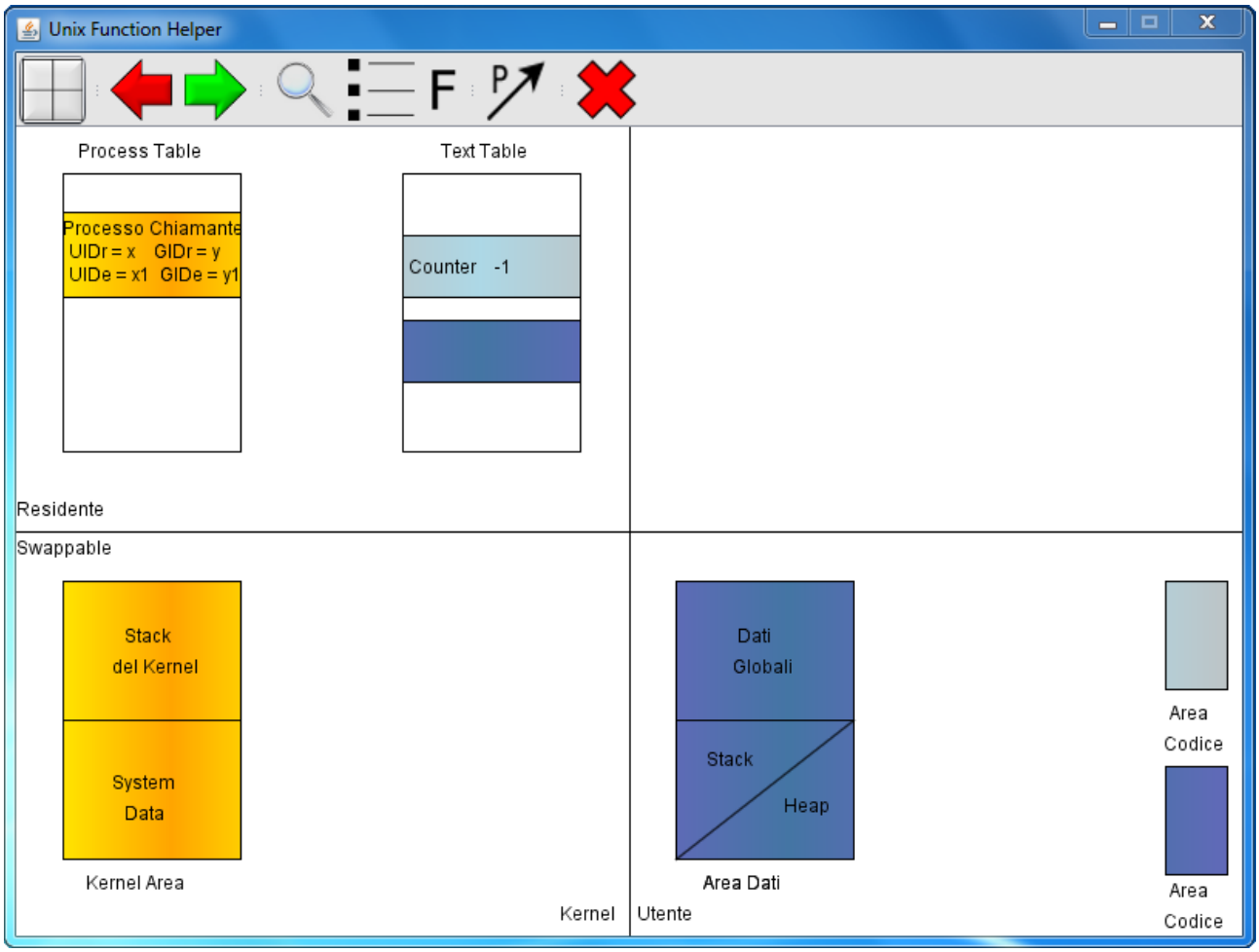

Figura 5.10 Situazione dopo l'esecuzione della primitiva Exec senza la combinazione con la Fork

Quello che si può notare è un cambiamento del colore dell'area Dati del processo chiamante in linea con la sua nuova area di codice e la nuova entry nella Text Table alla quale il processo chiamante farà riferimento per raggiungere l'area del codice.

Sfruttando il bottone della Fork, si ha un'immagine che risulta rappresentare la medesima situazione della Figura 5.4, senza la scritta Counter +1 perchè viene raffigurato il punto di partenza.
Quindi viene riportato solamente l'effetto della primitiva Exec sul processo figlio quando questa è stata eseguita (Figura 5.11)

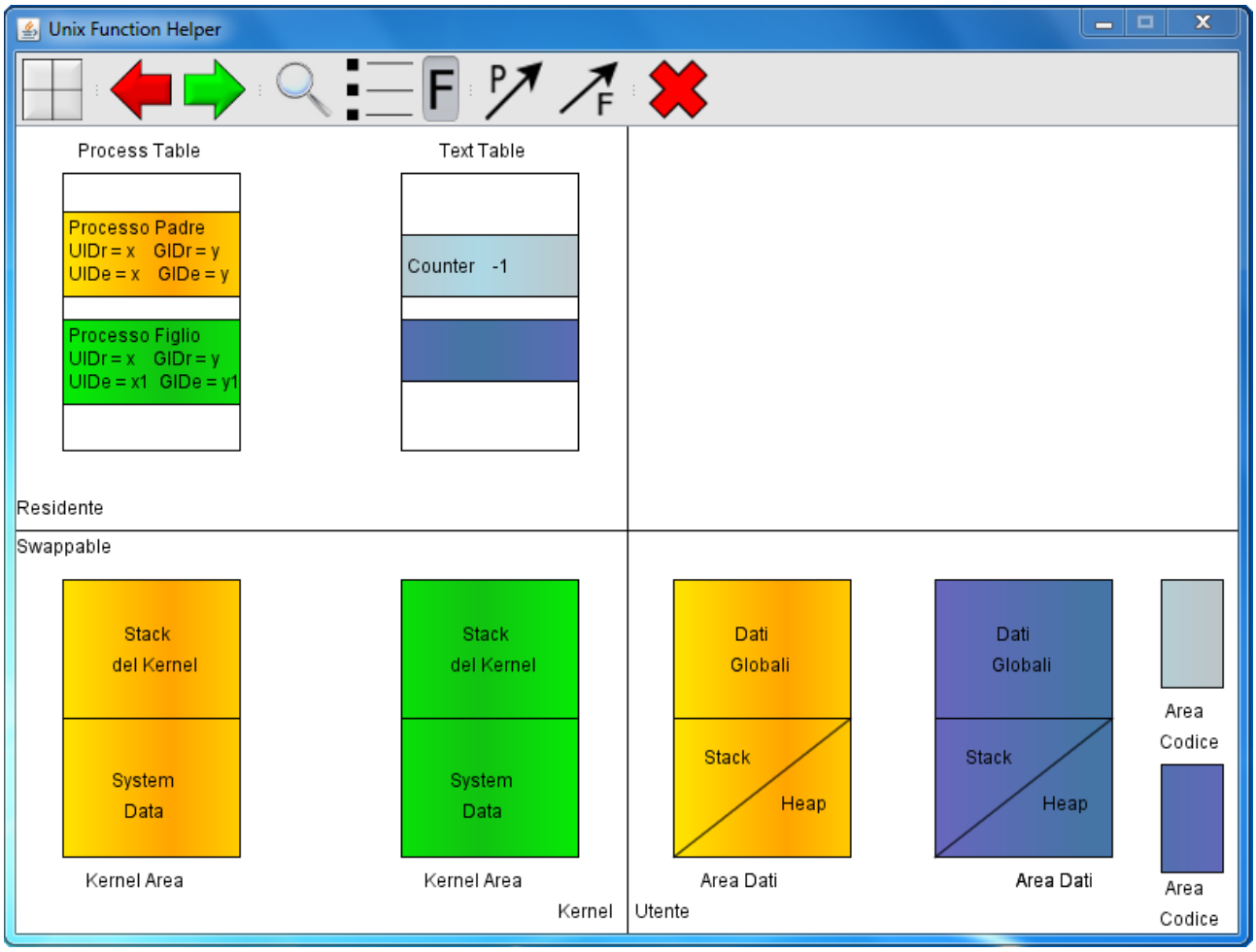

Figura 5.11 Effetto della primitiva Exec sfruttando la combinazione con la Fork

Le modifiche effettuate sul processo chiamante nella figura precedente sono ora applicate al processo figlio.

La possibilità di combinare 2 primitive esiste solo per la primitiva Exec all'interno di questa applicazione.

5.2.3 Le primitive Open e Creat

Per quanto riguarda le primitive Open e Creat, il disegno tratta tabelle diverse da quelle raffigurate nelle figure precedenti, eccezion fatta per l'opzione sul File all'interno della primitiva Fork, che rappresenta proprio l'insieme di componenti sui quali queste 2 primitive agiscono.
In Figura 5.12 viene mostrata la situazione di partenza per queste 2 primitive. Per quanto riguarda la Open, effettuando l'apertura del File, essa costruirà un'immagine completamente uguale alla raffigurazione presente nella Fork prima dell'azione della primitiva e con il bottone File selezionato.
Quindi si può affermare che gli effetti della Open su queste tabelle si possono osservare nella Figura 5.8.

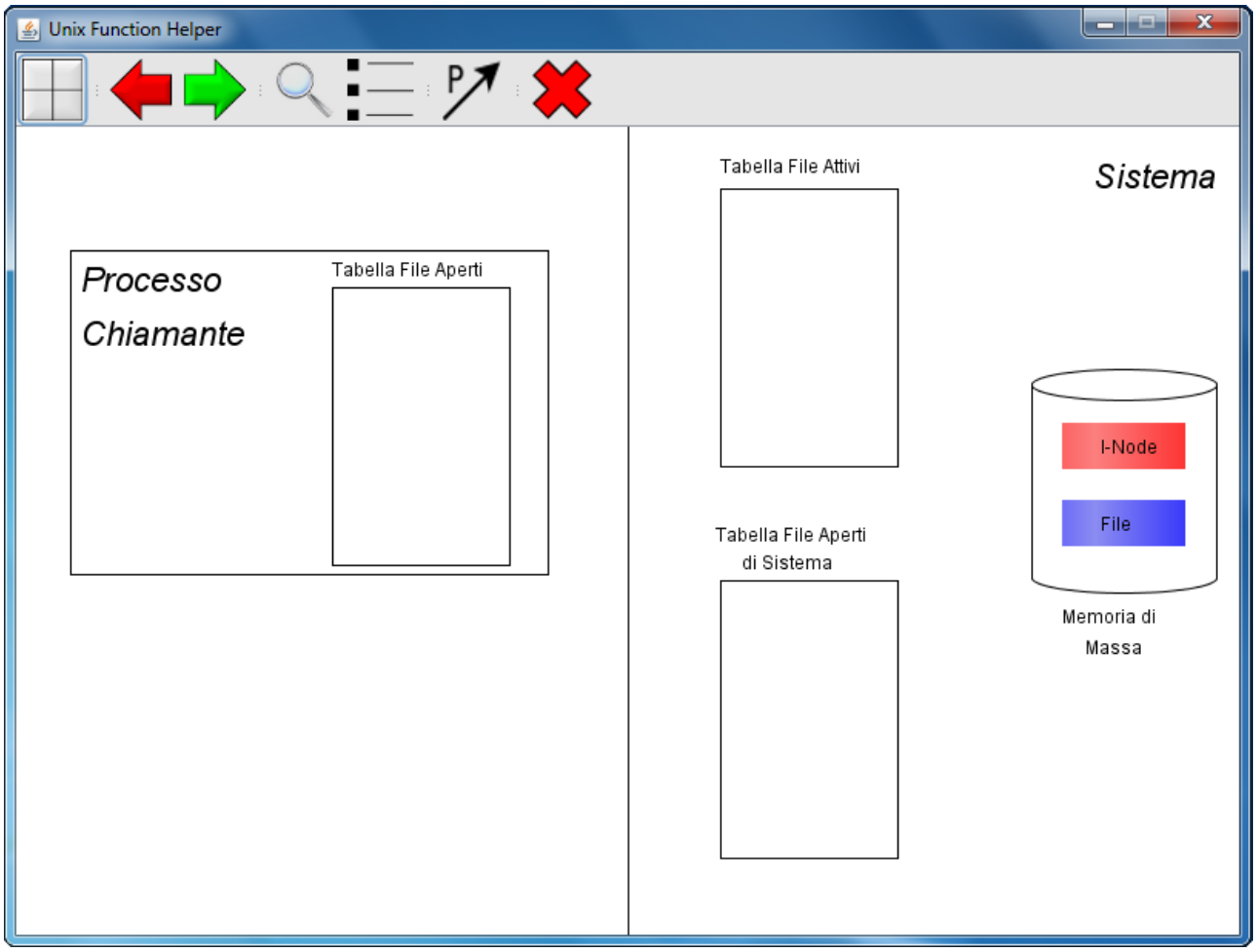

Figura 5.12 Situazione di partenza per la Open e per la Creat

Anche per la primitiva Creat si può considerare l'immagine qui presente come situazione di partenza, ma occorre fare una piccola precisazione: dal momento che la Creat è in grado di creare un appunto un file, l'utente troverà questa rappresentazione solo se avrà selezionato il bottone File all'interno della Creat stessa, ovvero se il File risulta essere già esistente in Memoria di Massa.
Se il File non esiste la rappresentazione è molto simile alla Figura 5.12, eccetto che per la Memoria di Massa vuota.

Gli effetti della Creat sono simili, dal punto di vista grafico, a quelli della Open, eccetto la differente modalità di apertura del file che per la Creat è sempre solo in scrittura.
Se il file esiste all'interno della Memoria di Massa quando viene chiamata questa primitiva, allora esso viene azzerato: nella Figura 5.13 si può osservare questa caratteristica nella differente colorazione, rispetto alle altre immagini, del blocco relativo al file.

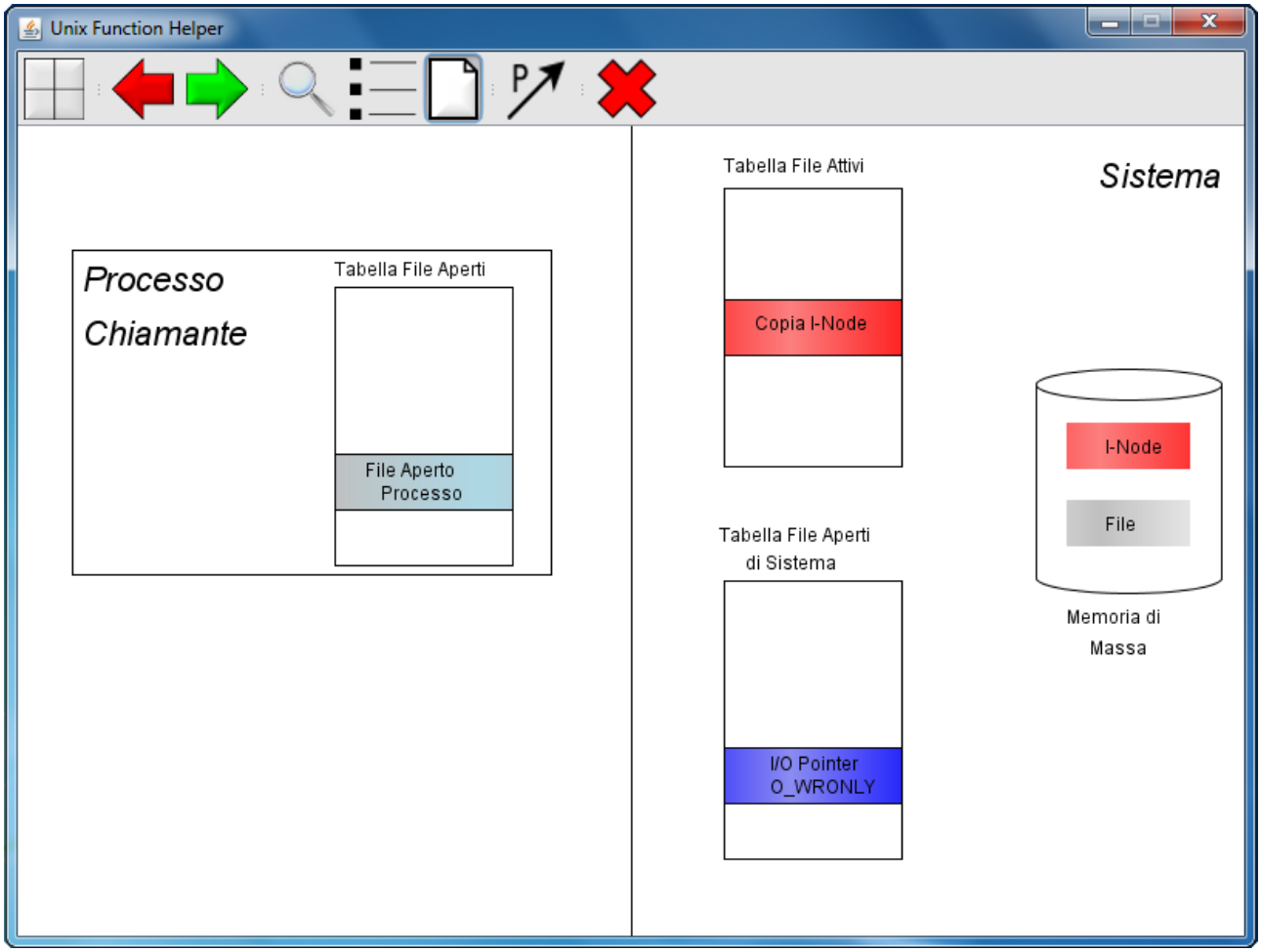

Figura 5.13 Effetti della Creat con File preesistente in Memoria di Massa

Conclusioni

In questo elaborato è stata analizzata nel completo l'applicazione Unix Function Helper, sia dal punto di vista delle ragioni che hanno portato alla sua progettazione sia dal punto di vista implementativo.
Durante lo sviluppo del programma si è acquisita una maggiore padronanza del linguaggio Java, sicuramente nell'ambito della realizzazione di interfacce utente e del disegno bidimensionale tramite l'utilizzo delle librerie Swing e AWT e lo studio delle caratteristiche grafiche del linguaggio stesso. Inoltre sono stati considerati alcuni aspetti della programmazione concorrente, in particolare il legame tra i concetto di thread e quello di reattività dell'interfaccia.
Alcuni sviluppi futuri per l'applicazione potrebbero essere:

1) ampliare il numero delle primitive supportate, per presentare allo studente uno strumento più completo;
2) combinare tra loro gli effetti delle varie primitive, come mostrato dal punto di vista della Fork e dell' Exec, in modo da comprendere il maggior numero di casi possibili che si possono verificare in ambiente Unix;
3) eseguire una programmazione più spinta dal punto di vista grafico per rendere l'interfaccia ancora più utile all'utente, garantendo sempre più funzionalità.

Bibliografia

Pellegrino Principe, *Java 7,* 2011

API Java

http://docs.oracle.com/javase/6/docs/api/

Tutorial sul disegno 2D in Java

http://docs.oracle.com/javase/tutorial/2d/

Sito web del corso di Ingegneria del Software

http://pervasive2.morselli.unimo.it/~nicola/courses/IngegneriaDelSoftware/

Lucidi dell'insegnamento di Sistemi Operativi e Lab.

http://agentgroup.unimo.it/wiki/index.php/Sistemi_Operativi_e_Lab.#Lucidi_dell.27Insegnamento