Università degli Studi di Roma
Tor Vergata

Corso di Laurea Specialistica in Ingegneria Informatica

Computer Viruses and the Simulation Environment WiCE

Relatore
Prof. Alberto Pettorossi

Studente
Fernando Iazeolla

Anno Accademico 2006/2007

.

Abstract

There has been considerable interest in computer viruses since they first appeared in 1981, and especially in the past few years as they have reached epidemic numbers in many personal computer environments. Viruses have been written about as a security problem, as a social problem, and as a possible means of performing useful tasks in a distributed computing environment.
However, only recently some scientists have begun to ask if computer viruses are not a form of artificial life; a self-replicating organism. Simply because computer viruses do not exist as organic molecules may not be sufficient reason to dismiss the classification of this form of "vandal-ware" as a form of life.
This thesis begins with a formal description of what a computer virus is. An abstract theory is presented. It covers an operational semantic of computer viruses in term of statements and machine status, and a denotational semantic, first presented by Adleman, in which is computer viruses are seen in a much higher level abstraction.
Real computer viruses are classified according to their infection methods, environments upon which they depend on, and in-memory strategies. Then we discuss advanced techniques used by computer viruses such as anti-disassembling, anti-debugging, encrypting, and polymorphism.
The thesis ends with the implementation of a simulating environment WiCE. The WiCE environment is able to run two or more self-modifying codes in a "sand-box" (a protected environment) that separates the viral codes from the real machine.

.

Dedicato ai miei genitori...

Life, Love and Liberation.

November 6, 2007

Acknowledgments

Prof. A.D.Pettorossi.
Alberto, Silvia, Guglielmo, Barbara, Beatrice, Gabriele, Mauro, Paola, Raymond, Raoul, Eva, Claudia, Alessanra, Rosa, Mario.
and... (in alphabetical order):
Francesco Agrusti, Alessandro Alicata, Carlo Alicata, Viviana Bessone, Antonio Campagna, Pamela Campi, Francesca Cartia, Marco Carta, Francesco Caputo, Daniela Caruso, Gianni Ceneri, Luigi Ceneri, Andrea Ciarletta, Alessio Ciancio, Giulio Comerci, Luca Conticelli, Fabio D'Andria, Marco Dell'aquila, Luigi Del Prete, Mauro Di Gennaro, Daniele Di Pinto, Filippo Ercoli, Marco Lattanzi, Valentina Fornaiolo, Laura Frezza, Stefano Gallo, Nazzareno Iannucci, Daria Ludovci, Ilenia Miduri, Elena Macri', Michele Martone, Alessandro Monterosso, Giuliano Monterosso, Alessia Paradisi, Francesco Pedini, Flavio Pierri, Roberto Pizzotti, Alessandro Polselli, Francesco Pozzi, Luciano Raguzzi, Riccardo Raguzzi, Veronica Rao, Filippo Restanti, Pierpaolo Rospo, Silvia Rossi, Sebastiano Russo, Massimiliano Saccone, Paolo Salvatore, Simona Sgro', Fabrizio Sisti, Carlo Spaccasassi, Stefano Tonna, Giuseppe Valletta, Lidia Vastola, Maria Clara Vendittelli, Chiara Visca, Federico Zannini, Pierluigi Zappalenti, Massimiliano Zenti... and all those i forgot who supported and encouraged me.
Special thanks to *the infinite monkey theorem* && "cat /dev/random".
Thanks also to Daniel (Life, Love and Liberation), Daniele Silvestri (Il latitante) and Niccolo' Fabi (la cura del tempo, dischi volanti) whose music i listened to while i was coding... the soundtrack of my thesis.
Special thanks to *the Death Valley (Nevada, USA)*. To rebirth into a new life, you first have to die. There is no place better in the world to die than the death valley.

Contents

Part I

Computer Viruses

Chapter 1

Introduction

1.1 Genesis of Computer Viruses

In 1949, John von Neumann devises the theory of self-replicating programs (see 2.2), providing the theoretical foundation for computers that hold information in their "memory".
Virus-like programs appeared on microcomputers in the 1980s. However, two fairly recounted precursors deserve mention here: Creeper from 1971-72 and John Walker's "infective" version of the popular ANIMAL game for UNIVAC in 1975.
Creeper and its nemesis, Reaper, the first "antivirus" for networked TENEX running on PDP-10s at BBN, was born while they were doing the early development of what became "the internet".
Even more interestingly, ANIMAL was created on a UNIVAC 110/42 mainframe computer running under the Univac 1100 series operating system, Exec-8. In January of 1975, John Walker (later founder of Autodesk, Inc. and co-author of AutoCAD) created a general subroutine called PERVADE, which could be called by any program. When PERVADE was called ANIMAL, it looked around for all accessible directories and made a copy of its caller program, ANIMAL in this case, to each directory to which the user had access.
The first virus (Elk Cloner) on microcomputers was written on the apple][, circa 1982. Elk Cloner had a payload that displayed Skrenta's poem after every 50^{th} use of the infected disk when reset was pressed. On every 50^{th} boot, Elk Cloner hooked the reset handler; thus only pressing reset triggered the payload of the virus.
The first IBM-PC virus appeared in 1986; this was the Brain virus. Brain was a boot sector virus and remained resident. In 1987, Brain was followed by Alameda (Yale), Cascade, Jerusalem, Lehigh, and Miami (South African Friday the 13th). These viruses expanded the target executables to include COM and EXE files. Cascade was encrypted to deter disassembly and detection. Variable encryption appeared in 1989 with the 1260 virus. Stealth viruses, which employ various techniques to avoid detection, also first appeared in 1989, such as Zero Bug, Dark Avenger and Frodo

(4096 or 4K). In 1990, self-modifying viruses, such as Whale were introduced. The year 1991 brought the GP1 virus, which is "network-sensitive" and attempts to steal Novell NetWare passwords. Since their inception, viruses have become increasingly complex. Examples from the IBM-PC family of viruses indicate that the most commonly detected viruses vary according to continent, but Stoned, Brain, Cascade, and members of the Jerusalem family, have spread widely and continue to appear. This implies that highly survivable viruses tend to be benign, replicate many times before activation, or are somewhat innovative, utilizing some technique never used before in a virus.
Internet and e-mail revolutionized communications. However, as expected, virus creators didn't take long to realize that along with this new means of communication, an excellent way of spreading their creations far and wide had also dawned. Therefore, they quickly changed their aim from infecting a few computers while drawing as much attention to themselves as possible, to damaging as many computers as possible, as quickly as possible. This change in strategy resulted in the first global virus epidemic, which was caused by the Melissa worm.

Chapter 2

An Abstract Theory of Computer Viruses

2.1 Turing Machines

Alan Turing in 1936 introduced an abstract model for computation called Turing Machine. There are various variants of this model, which can be proved to be equivalent in the sense that they are all capable to compute the same set of functions from N to N.

Informally, we can say that a *Turing Machine* (TM, for short) M consist of: (i) a *finite automaton* FA, also called *control*, (ii) a one-way infinite tape divided into *cells* $\{c_i | i \in N, i > 0\}$, and (iii) a tape head which is on a cell at a time, called the *scanned cell*. Each cell contains exactly one of the symbols of the *Tape alphabet* Γ. The states of the FA are also called *internal states* (or simply *states*), of the Turing Machine M. We assume a left-to-right direction on the tape by stipulating that for $i > 0$ the cell c_i is immediately to the left of the cell c_{i+1}.

A Turing Machine behaves as follows. It starts on a tape containing in its cells $c_1 c_2 \dots c_n$ a sequence of n input symbols from the *input alphabet* Σ, while all other cells contain the symbol B (called *blank*) belonging to Γ. We assume that: $\Sigma \subseteq \Gamma - \{B\}$. The Turing Machine M starts with its tape head on the leftmost cell, that is c_1, and the FA in its initial state q_0.

A *move* (or a *transition*) of the TM is given by a quintuple:

$$q_i, X_h \rightarrow q_i, X_k, m$$

where: (i) q_i is the current state of the FA, (ii) X_h is the symbol on the scanned cell (that is the symbol which is read by the tape head), (iii) q_j is the new state of the FA, (iv) X_k is the symbol which ,after the move, replaces X_h on the scanned cell (X_k is also called the printed symbol), and (v) m is either L or R and denotes that the tape head, after the move, will be on the cell to the left or to the right, respectively, of the cell scanned before the move.

No two quintuples have the same first two components, and we refer to this property by saying that the TM is *deterministic*.

Definition 2.1.1. *A Turing Machine (or a TM, for short) is a septuple* $\langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle$, *where:*

- Q *is the set of states,*
- Σ *is the input alphabet,*
- Γ *is the tape alphabet,*
- δ *is a partial function from* $Q \times \Gamma$ *to* $Q \times (\Gamma - \{B\}) \times \{L, R\}$, *called transition function, which defines the set of quintuples of the Turing Machine,*
- q_0 *is the initial state,*
- B *is the blank symbol, and*
- F *is the set of the final states.*

We assume that Q and Γ are disjoint, and $\Sigma \subseteq \Gamma - \{B\}$.

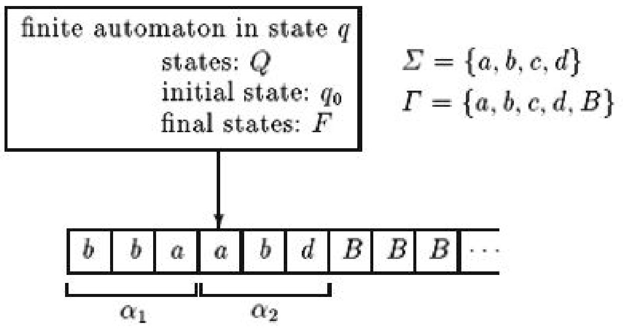

finite automaton in state q
states: Q
initial state: q_0
final states: F
$\Sigma = \{a, b, c, d\}$
$\Gamma = \{a, b, c, d, B\}$
b b a a b d B B B ...
α_1
α_2

Figure 2.1: a Turing Machine in the configuration $\alpha_1\, q\, \alpha_2$, that is, $b\, b\, a\, q\, a\, b\, d$

.

Definition 2.1.2. *A configuration of a TM whose tape head is reading the cell* c_h *for some* $h \geq 1$, *is the triple* $\alpha_1\, q\, \alpha_2$, *where:*

- α_1 *is the (possibly empty sequence of* $(\Gamma - \{B\})^{h-1}$ *contained in the cells* $c_1 c_2 \ldots c_{h-1}$,
- q *is the current state of the TM, and*

- α_2 *is the non-empty sequence of* Γ^{k-h+1} *contained in the cells* $c_h \ldots c_k$*, if the tape head is reading a non-blank symbol (that i,h* $\leq k$*). Otherwise,* α_2*, is the sequence of one* B *only, if the tape head is reading a blank symbol* B *(that is,* $h = k + 1$*).*

For any given configuration we assume that the tape head is reading the leftmost symbol of α_2.

2.2 von Neumann's Theory of Self-Reproducing Automata

Replication is an essential part of life. John von Neumann was the first to provide a model to describe nature's self-reproduction [NEUMNN] with the idea of self-building automata.
In von Neunamm's vision, there were three main components in a system:

1. A Universal Machine,

2. A Universal Constructor,

3. Information on a Tape.

A universal machine (Turing Machine) would read the memory tape and, using the information on the tape, it would be able to rebuild itself piece by piece using a universal constructor.The machine would not understand the process. It would simply follow the information (blueprint instructions) on the memory tape. The machine would only be able to select the next proper piece from the set of all the pieces by picking them one by one until the proper piece was found. When it was found, two proper pieces would be put together according to the instructions until the machine reproduced itself completely.
If the information that was necessary to rebuild another system could be found in the tape, then the automata was able to reproduce itself. The original automata would be rebuilt (Figure 2.2), and then the newly built automata was booted, which would start the same process.

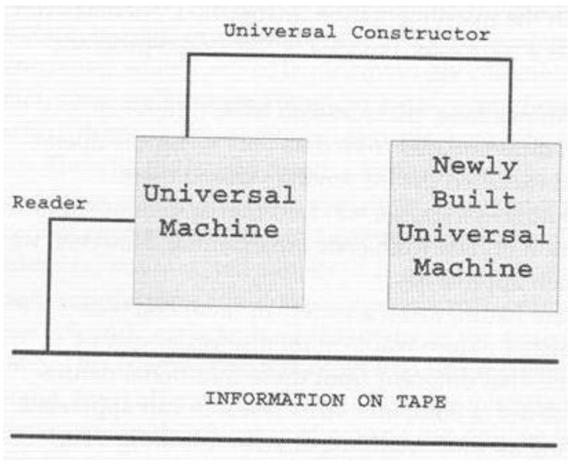


Figure 2.2: The model of a self-reproducing machine.

A few years later, Stanislaw Ulam suggested to von Neumann to use the processes of cellular automation to describe those model. Instead of using "machine parts", states of cells were introduced. Because cells are operated in robotic fashion according to rules ("code"), the cell is known as an automation. The array of cells comprises the *cellular automata* (CA) computer architecture.
In 1948 von Neumann presented his vision of self-replicating automata systems. Only five years later, in 1953, Watson and Crick recognized that living organisms use the DNA molecule as a "tape" that provides the information for the reproduction system of living organisms.
Unfortunately, von Neumann could not see a proof of his work in his life, but his work was completed by Arthur Burks.

2.3 Computational Domains

2.3.1 Complete Partial Order (CPO)

Definition 2.3.1. *A partial order is a set P on which is defined a binary relation $\sqsubseteq$ that have the following properties:*

1. *reflexive:* $\forall p \in P.\ p \sqsubseteq p$
2. *transitive:* $\forall p, q, r \in P.\ p \sqsubseteq q \ \&\ q \sqsubseteq r \Rightarrow p \sqsubseteq r$
3. *antisimmetric:* $\forall p, q \in P.\ p \sqsubseteq q \ \&\ q \sqsubseteq p \Rightarrow p = q$

Definition 2.3.2. *Given a partial order (P,$\sqsubseteq$) and a subset $X \subseteq P$, p is an upper bound of X iff*

$$\forall q \in X.\ q \sqsubseteq p.$$

We say that p is a least upper bound (for short, lub) of X iff

1. *p is an upper bound of X, and*
2. *for all upper bound q of X, $p \sqsubseteq q$.*

The least upper bound in indicated as $\bigsqcup X$.

Definition 2.3.3. *Given a partial order (D,$\sqsubseteq$), a ω-chain of the partial order is an increasing chain $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \cdots \sqsubseteq_D \cdots$ elements of the partial order.*
A partial order (D,$\sqsubseteq$) is a complete partial order (for short, cpo) if all of its ω-chains $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \cdots \sqsubseteq_D \cdots$ have a lub, that is, every increasing chain $\{d_n | n \in \omega\}$ of elements of D has an upper bound $\bigsqcup\{d_n | n \in \omega\}$ in D. This may be indicated with: $\bigsqcup_{n\in\omega} d_n$.

An imperative programming language (for short, IMP) is made of arithmetical expressions, boolean expressions and commands. Each of those have his proper domains:

- arithmetical expressions $A: \; Aexp \rightarrow (\Sigma \rightarrow N)$
- boolean expressions $B: \; Bexp \rightarrow (\Sigma \rightarrow T)$
- commands $C: \; Com \rightarrow (\Sigma \rightharpoonup \Sigma)$

where N is the set of natural numbers, T is the set (true,false), and Σ is the set of all possible states.
A particular state is the *bottom* state (denoted as $\perp$)(see 2.3.2). This state is a result of a non-terminated command (for example the *'while(true) { }'* command). So as winskel says [WNSKL], the command domain is always defined because we have defined the non-termination state as a normal and possible state. And now the notation of denotational semantic is more easy. The set of possible states is now extended (in the cpo $\Sigma_\perp$) by adding a new element that is the least of the set:

$$\forall \sigma \in \Sigma. \;\; \perp \sqsubseteq \sigma$$

The partial functions $\Sigma \rightharpoonup \Sigma$ are in one-to-one correspondence with the total functions $\Sigma \rightarrow \Sigma$.
The complete partial orders are data types that can be used as input or output of a computation. So the domains of $Aexpr, Bexpr$ and Com are CPOs. The CPO's elements may be seen as information points, and in the relation $x \sqsubseteq y$, x approximates y (or x contains minor information than y). Consequently $\perp$ is the minimum (null) information point.

2.3.2 Lifting

The operation of adding the $\perp$ element to a CPO is known as *lifting.* Informally, the lifting process adds a new element that is minor of all other elements of the CPO, to a copy of the original CPO.
Assume D a CPO. the lifting process assume the existence of an element $\perp$ and a function $\lfloor - \rfloor$ with the following properties:

- $\lfloor d_0 \rfloor = \lfloor d_1 \rfloor \Rightarrow d_0 = d_1$, and
- $\perp \neq \lfloor d \rfloor$

for all $d, d_0, d_1 \ \in D$. The set of the elements of the $D_\perp$ CPO obtained from the lifting process is defined as follows:

$$D_\perp = \{\lfloor d \rfloor \mid d \in D\} \cup \{\perp\},$$

and the partial order is:

$$d_0^{'} \sqsubseteq d_1^{'} \text{ iff } \begin{cases} d_0^{'} = \perp & \text{or} \\ \exists d_0, d_1 \in D.\ d_0^{'} = \lfloor d_0 \rfloor \&\ d_1^{'} = \lfloor d_1 \rfloor \&\ d_0 \sqsubseteq_D d_1. \end{cases}$$

So $\lfloor d_0 \rfloor \sqsubseteq \lfloor d_1 \rfloor$ in $D_\perp$ only if $d_0 \sqsubseteq d_1$. And so $D_\perp$ is a copy of the D CPO with a new minimum element $\perp$ that is distinct from all the others (Figure 2.3.2).

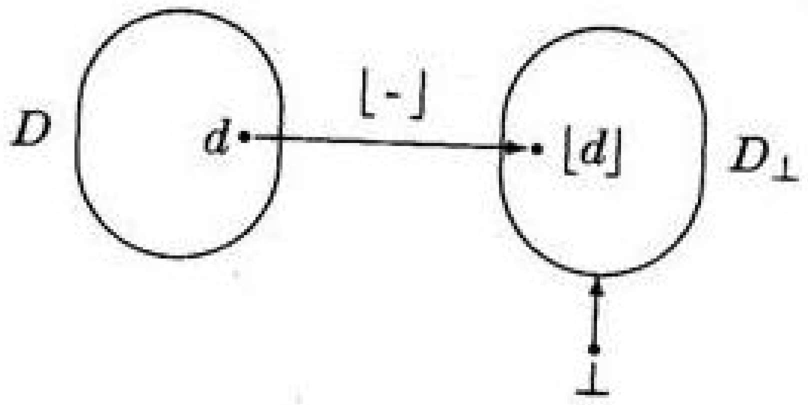


Figure 2.3: Graphic representation of the lifting operation

2.4 Operational Semantic of IMP

As known the execution of a command can bring to a final state or stay "bottom" and never reach a final state. With the notation $\langle c, \sigma \rangle$ we denote a command configuration that means that we are to execute the command c from the state σ. When the execution stops, it ends in a final state $\sigma^{'}$

$$\langle c, \sigma \rangle \to \sigma^{'}$$

For example

$$\langle X := 5, \sigma \rangle \to \sigma^{'}$$

means that the state $\sigma^{'}$ is obtained from the state σ updating the location X with the number 5.
Here is the syntax of the IMP language:

- *Aexpr* : $a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$.
- *Bexpr* : $b ::= \text{true} \mid \text{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$.
- *Com* : $c ::= \text{skip} \mid X := a \mid \text{if } b \text{ then } c_0 \text{ else } c_1 \mid \text{ while } b \text{ do } c$.

We leave the rules of *Aexpr* and *Bexpr* to the reader [WNSKL], and give the rules of the operational semantic of commands:

$$\langle \text{skip}, \sigma \rangle \to \sigma \tag{2.1}$$

$$\frac{\langle a, \sigma \rangle \to m}{\langle X := a, \sigma \rangle \to \sigma[m/X]} \tag{2.2}$$

$$\frac{\langle c_0, \sigma \rangle \to \sigma^{''} \ \langle c_1, \sigma^{''} \rangle \to \sigma^{'}}{\langle c_0; c_1, \sigma \rangle \to \sigma^{'}} \tag{2.3}$$

$$\frac{\langle b, \sigma \rangle \to \text{true} \ \langle c_0, \sigma \rangle \to \sigma^{'}}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \to \sigma^{'}} \tag{2.4}$$

$$\frac{\langle b, \sigma \rangle \to \text{false} \ \langle c_1, \sigma \rangle \to \sigma^{'}}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \to \sigma^{'}} \tag{2.5}$$

$$\frac{\langle b, \sigma \rangle \to \text{false}}{\langle \text{while } b \text{ do } c, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to \text{true} \ \langle c, \sigma \rangle \to \sigma^{''} \ \langle \text{ while } b \text{ do } c, \sigma^{''} \rangle \to \sigma^{'}}{\langle \text{while } b \text{ do } c, \sigma \rangle \to \sigma^{'}} \tag{2.6}$$

Even on command is defined the equivalence relationship:

$$c_0 \sim c_1 \text{ iff } \forall \sigma, \sigma^{'} \in \Sigma. \ \langle c_0, \sigma \rangle \to \sigma^{'} \iff \langle c_1, \sigma \rangle \to \sigma^{'} \tag{2.7}$$

2.5 Basic Virus Definitions

For the purpose of motivating the definitions which follows, consider this 'case study':

A text editor becomes infected with a computer virus. Each time the text editor is used, it performs the text editing tasks as it did prior to infection, but it also searches the files for a program and infects it. When run, each of these newly infected programs performs its 'intended' tasks as before, but also searches the files for a program and infects it. This process continues. As these infected programs pass between systems, as when they are sold, or given to others, new opportunities for spreading the virus are created. Finally, after Jan. 1, 2010, the infected programs cease acting as before. Now each time such program is run, it deletes all files.

Such a "classic" computer virus can follows a scheme similar to this:

```
{ main:=
call injure;
...
call submain;
...
call infect;
}
{ injure:=
if 'condition' then whatever damage is to be done and halt
}

{ infect:=
if 'condition' then infect files
}
```

By modifying the scheme above, a wide variety of "classic" viruses can be created. From the above scheme, it appears that the following properties are relevant:

1. For every program, there is an 'infected' form of that program. That is, it is possible to think of the virus as a function that maps programs to ('infected' form) programs.
2. Each infected program on each input (where by input is meant all 'accessible' information such as the user input, or the system clock, or files containing data or programs) makes one of tree choices:

Injure: ignore the 'intended' task and compute some other function. Note that which inputs result in injury and what kind of injury occurs are the same whether the infected program is a text editor or a compiler or something else.Thus which inputs result in injury and what form the injury takes is independent of which infected program is running and is actually dependent solely on the virus itself.

Infect: Perform the 'intended' task and if it halts, infect programs. Notice that all the resource such as clock, the user/program communications and all 'accessible' information other than programs, are handled just as they would have been had the uninfected version of the program been run. Further, notice that whether the infected program is a text editor or a compiler or something else, when it infects a program the resulting infected program is the same. Thus the infected form of a program is independent of which infected program produces the infection.

Imitate: Neither injure nor infect. Perform the 'intended' task without modification. This may be thought of as a special case of 'Infect', where the number of programs getting infected is zero. (in the case of study, imitation only occurs when no programs are accessible for infection).

In more sophisticated viruses the injury process may not halt when finished (as they do in classical virus), but an 'Imitate' process can be done. This is done to hide as long as possible the virus to the user.(e.g., an injury procedure that deletes disk tracks or sector randomly then continues to preform the original program task as 'intended').

2.5.1 An Operational Semantic of Computer Viruses

A program $P : \Sigma \to \Sigma_\perp$ is intended as a sequence (equation 2.3) of commands.

$$P = c_0; c_1; \dots; c_z = C[c_0; c_1; \dots; c_z] = C[c_z] \circ \cdots \circ C[c_2] \circ C[c_1] \tag{2.8}$$

where z is the length of the program.

P_1 is the "clean" program, P_2 is the infected form of the program. Both are in the form shown in Eq.2.8

Definition 2.5.1. *(clean)* $P_1 = P_2$ *iff*

$$\forall i. 0 \leq i \leq z,\; c_{i_{p_1}} = c_{i_{p_2}}.$$

Definition 2.5.2. *(imitate)* $P_1 \simeq P_2$ *iff*

$$X \subseteq \Sigma\,,\; \forall \sigma \in \Sigma\,,\; \forall x^{'}, x^{''} \in X.(\langle P_1, \sigma \rangle \to x^{'} \;\&\; \langle P_2, \sigma \rangle \to x^{''}) \Rightarrow x^{'} = x^{''}.$$

where X is the set of all "visible states" (e.g., output). Visible sates are a subset of the machine states Σ that a user can see (altered).

Definition 2.5.3. *(infect)* $P_1 \overset{h}{\sim} P_2$ *iff*

$$\exists i. c_{i_{p_1}} \neq c_{i_{p_2}} \ \&$$
$$\forall i.\ 0 \leq i \leq z \ \textit{either} \begin{cases} c_{i_{p_1}} = c_{i_{p_2}} \ \textit{or} \\ h(c_{i_{p_1}}) = c_{i_{p_2}}. \end{cases}$$

Definition 2.5.4. *(infect or imitate)* $P_1 \overset{h}{\cong} P_2$ *iff* $P_1 \simeq P_2$ *or* $P_1 \overset{h}{\sim} P_2$

Definition 2.5.5. *(injury)* $P_1 \neq P_2$ *iff*

$$P_1 \overset{h}{\sim} P_2 \ \&$$
$$\forall \sigma, \sigma' \in \Sigma.(\langle P_1, \sigma \rangle \to \sigma^{'} \ \& \ \langle P_2, \sigma \rangle \to \sigma^{''}) \Rightarrow \sigma^{'} \neq \sigma^{''}$$

Definition 2.5.6. *A program is* ***pathogenic*** *iff*

$$P_1 \ncong_h P_2 \ \&$$
$$P_1 \neq P_2$$

Definition 2.5.7. *A program is* ***contagious*** *iff*

$$P_1 \sim_h P_2$$

Definition 2.5.8. *A program is* ***benignant*** *iff*

- *is not pathogenic*
- *is not contagious*

Definition 2.5.9. *A program is a* ***Trojan horse*** *iff*

- *is pathogenic*
- *is not contagious*

Definition 2.5.10. *A program is a* ***carrier*** *iff*

- *is not pathogenic*
- *is contagious*

Definition 2.5.11. *A program is* ***virulent*** *iff*

- *is pathogenic*
- *is contagious*

2.5.2 A Denotational Semantic of Computer Viruses

In the previous section we have seen the basic virus definition from an operational point of view. So we intended a program as a sequence of commands (statements) and the input as a set of state that can be modified by the program (output).
In this section we want to present another point of view: a functional point of view. A program s a partial recursive function from the set of natural numbers to its self (input/output). This theory was first introduced by Adleman [VIRTHRY] in early 1980s.

Definition 2.5.12. *1. S denotes the set of all finite sequences of natural numbers.*

2. *e denotes a computable injective function from $S \times S$ onto N with computable inverse.*
3. *$\forall s, t \in S$, $\langle s, t\rangle$ denotes $e(s, t)$*
4. *For all partial $f : N \rightarrow N$, $\forall s, t \in S$, $f(s, t)$ denotes $f(\langle s, t\rangle)$.*
5. *$e^{'}$ denotes a computable injective function from $N \times N$ onto N with computable inverse such that $\forall i, j \in N$, $e^{'}(i, j) \geq i$.*
6. *$\forall i, j \in N$, $\langle i, j\rangle$ denotes $e^{'}(i, j)$.*
7. *For all partial $f : N \rightarrow N$, $\forall i, j \in N$, $f(i, j)$ denotes $f(\langle i, j\rangle)$.*
8. *For all partial $f : N \rightarrow N$, $\forall n \in N$, write $f(n) \downarrow$ iff $f(n)$ is defined.*
9. *For all partial $f : N \rightarrow N$, $\forall n \in N$, write $f(n) \uparrow$ iff $f(n)$ is undefined.*

Definition 2.5.13. *For all partial $f, g : N \rightarrow N$, $\forall s, t \in S$, $f(s, t) = g(s, t)$ iff either:*

1. *$f(s, t) \uparrow$ & $g(s, t) \uparrow$ or*
2. *$f(s, t) \downarrow$ & $g(s, t) \downarrow$ & $f(s, t) = g(s, t)$*

Definition 2.5.14. *For all $z, z' \in N$, $\forall p, p', q = q_1, q_2, \ldots, q_z$, $q' = q'_1, q'_2, \ldots, q'_z \in S$, for all partial functions $h : N \rightarrow N$, $\langle p, q\rangle \overset{h}{\sim} \langle p', q'\rangle$ iff:*

1. *$z = z'$ and*
2. *$p = p'$ and*
3. *there exists an i, with $1 \leq i \leq z$ such that $q_i \neq q'_i$ and*
4. *for $i = 1, 2, \ldots, z$, either*
   (a) *$q_i = q'_i$ or*
   (b) *$h(q_i) \downarrow$ and $h(q_i) = q'_i$.*

Definition 2.5.15. *For all partial $f, g, h : N \to N$, $\forall s, t \in S$, $f(s,t) \overset{h}{\sim} g(s,t)$ iff $f(s,t)\downarrow$ & $g(s,t)\downarrow$ & $f(s,t) \overset{h}{\sim} g(s,t)$.*

Definition 2.5.16. *For all partial $f, g, h : N \to N$, $\forall s, t \in S$, $f(s,t) \overset{h}{\cong} g(s,t)$ iff $f(s,t) = g(s,t)$ or $f(s,t) \overset{h}{\sim} g(s,t)$.*

Definition 2.5.17. *For all Gödel numberings of the partial recursive functions $\{\phi_i\}$, a total recursive function v is a virus with respect to $\{\phi_i\}$ iff for all $d, p \in S$, either:*

1. *Injure:* $(\forall i, j \in N)[\phi_{v(i)}(d,p) = \phi_{v(j)}(d,p)]$
2. *Infect or Imitate:* $(\forall j \in N)[\phi_j(d,p) \overset{h}{\cong} \phi_{v(j)}(d,p)]$

Types of Viruses

In this section the set of viruses is decomposed into the disjoint union of four principal types.

Definition 2.5.18. *For all Gödel numberings of the partial recursive functions $\{\phi_i\}$, for all viruses v with respect to $\{\phi_i\}$, for all $i, j \in N$:*

- *i is pathogenic with respect to v and j iff*

$$\begin{gathered} i = v(j)\ \& \\ (\exists d, p \in S)[\phi_j(d,p) \overset{h}{\ncong} \phi_i(d,p)] \end{gathered}$$

- *i is contagious with respect to v and j iff*

$$\begin{gathered} i = v(j)\ \& \\ (\exists d, p \in S)[\phi_j(d,p) \overset{h}{\sim} \phi_i(d,p)] \end{gathered}$$

- *i is benignant with respect to v and j iff*

$$\begin{gathered} i = v(j)\ \& \\ i \text{ is not pathogenic with respect to } j\ \& \\ i \text{ is not contagious with respect to } j \end{gathered}$$

- *i is a Trojan horse with respect to v and j iff*

$$\begin{gathered} i = v(j)\ \& \\ i \text{ is pathogenic with respect to } j\ \& \\ i \text{ is not contagious with respect to } j \end{gathered}$$

- *i is a carrier with respect to v and j iff*

$$\begin{aligned}i = v(k)\ \&\\ i \text{ is not pathogenic with respect to } j\ \&\\ i \text{ is contagious with respect to } j\end{aligned}$$

- *$i$is virulent with respect to v and j iff*

$$\begin{aligned}i = v(j)\ \&\\ i \text{ is pathogenic with respect to } j\ \&\\ i \text{ is contagious with respect to } j\end{aligned}$$

Definition 2.5.19. *For all Gödel numberings of the partial recursive functions $\{\phi_i\}$, for all viruses v with respect to $\{\phi_i\}$:*

- *v is benign iff both:*
  - $(\forall j \in N)[v(j)$ *is not pathogenic with respect to v and j*$]$
  - $(\forall j \in N)[v(j)$ *is not contagious with respect to v and j*$]$
- *v is Epian iff both:*
  - $(\exists j \in N)[v(j)$ *is pathogenic with respect to v and j*$]$
  - $(\forall j \in N)[v(j)$ *is contagious with respect to v and j*$]$
- *v is disseminating iff both:*
  - $(\forall j \in N)[v(j)$ *is not pathogenic with respect to v and j*$]$
  - $(\exists j \in N)[v(j)$ *is contagious with respect to v and j*$]$
- *v is malicious iff both:*
  - $(\exists j \in N)[v(j)$ *is pathogenic with respect to v and j*$]$
  - $(\exists j \in N)[v(j)$ *is contagious with respect to v and j*$]$

Mutating Viruses (μ-viruses)

Definition 2.5.20. *For all $z, z' \in N$, for all $p, p', q = q_1, q_2, \ldots, q_z, q' = q'_1, q'_2, \ldots, q'_z \in S$, for all sets H of partial functions from N to N, $\langle p, q\rangle \overset{H}{\sim} \langle p', q'\rangle$ iff:*

1. *$z = z'$ and*
2. *$p = p'$ and*
3. *there exists an i, with $1 \leq i \leq z$ such that $q_i \neq q'_i$ and*

4. *for* $i = 1, 2, \ldots, z$ *either*
   (a) $q_i = q'_i$ *or*
   (b) *there exists an* $h \in H$ *such that* $h(q_i) \downarrow$ *and* $h(q_i) = q'_i$.

Definition 2.5.21. *For all sets of partial functions* H *from* N *to* N, *for all partial* $f, g : N \to N$, *for all* $s, t \in S$, $f(s,t) \overset{H}{\sim} g(s,t)$ *iff* $f(s,t) \downarrow$ & $g(s,t) \downarrow$ & $f(s,t) \overset{H}{\sim} g(s,t)$.

Definition 2.5.22. *For all sets of partial functions* H *from* N *to* N, *for all partial* $f, g : N \to N$, *for all* $s, t \in S$, $f(s,t) \overset{H}{\cong} g(s,t)$ *iff* $f(s,t) = g(s,t)$ *or* $f(s,t) \overset{H}{\sim} g(s,t)$.

Definition 2.5.23. *For all Gödel numberings of the partial recursive functions* $\{\phi_i\}$, *a set* M *of total recursive functions is a mutating virus,* μ*-virus, with respect to* $\{\phi_i\}$ *iff both:*

1. $\forall m \in M$, $\forall d, p \in S$ *either:*
   (a) *Injure:* $(\forall i, j \in N)[\phi_{m(i)}(d,p) = \phi_{m(j)}(d,p)]$
   (b) *Infect or Imitate:* $(\forall j \in N)[\phi_j(d,p) \overset{M}{\cong} \phi_{m(j)}(d,p)]$

Detecting The Set Of Viruses

theorem 2.5.24. *For all Gödel numberings of the partial recursive functions* $\{\phi_i\}$*:*

$$V = \{i | \phi_i \textit{ is a virus}\} \textit{ is } \Pi_2 \textit{ - complete}$$

Proof.
See Adleman's paper [VIRTHRY].

Chapter 3

Computer Architecture Dependency

One of the most important steps toward understanding computer viruses is learning about the particular environment in which they operate. In theory, for any given sequence of symbols we can define an environment in which that sequence could replicate itself. In practice, we need to be able to find the environment in which the sequence of symbols operates and prove that it uses code explicitly to make copies of itself and does so recursively.
A successful penetration of the system by viral code occurs only if the various dependencies of malicious code match a potential environment.
A virus may depend on a particular CPU (x86, PPC, Sparc) or on a particular Operating System. A virus that runs under MS-DOS can only run under Intel's 8086 CPU, earlier versions of Windows runs only on Intel x86 CPU, but Windows NT and newer runs even on MIPS, aplha and IA64. Linux runs on a lot of different hardware and different CPUs. If we consider high level languages or cross-platform languages such as Java and .NET, then the number of potential target of viruses covers almost every computer.
So a virus strictly depends on the environment (O.S.) it was designed for. If we consider the *Whale virus* (one of the earlier viruses) it would not replicate in modern computers because it has an interesting dependency on early 8088 architectures on which it works perfectly.
In theory, it would be feasible to create a multi-architecture binary virus (the *PeElf virus* in march 2001), but this is not a simple task.

3.1 CPU Dependency

The CPU dependency affects binary computer viruses. The source code of programs is compiled to object, which is linked in a binary format (executable format). The executable format contains the "genome" of a program as a sequence of instructions. The instructions consist of opcodes. Every CPU recognizes its own set of opcodes.

For instance, the instruction NOP (no operation) has different opcode on an Intel (0x90), on a VAX (0x01) and on a Machintosh PowerPC (there is no opcode for NOP, it can be simulated with the *ori 0,0,0* instruction which leaves unaltered the machine statuses).
RISC and CISC architecture has different opcodes format. In the RISC architecture an opcode takes always one word in which a group of bits have a particular meaning. On the contrary in CISC platform, an opcode can be made of one or more words.
There is yet another form of CPU dependency that occurs when a particular processor is not 100% backward compatible with the previous generation and does not support the features of another perfectly or at all. For example, the *Finnpoly virus* fails to work on 386 processors because the processor incorrectly executes the instruction "CALL SP" (make a call according to the Stack Pointer).
Some viruses use instructions that are simply no longer supported on newer CPU. For instance, the 8086 Intel CPU supported a "POP CS" instruction, although Intel did not document it. Later, the instruction opcode (0x0F) was used to trap into multiple opcode table. A similar example of this kind of dependency is the "MOV CS,AX" instruction used by some early computer viruses, such as the Italian boot virus, *Ping Pong*.
Other computer viruses might use the coprocessor or MMX (Multimedia Extensions) or some other extension, which is cause them to fail when they execute on a machine that does not support them.
Some viruses may use some kind of defense techniques based on altering the processor's prefetch queue. The size of the prefetch queue is different from processor to processor. Virus can try to overwrite code in the next instruction slot, hoping that such code is already in the processor prefetch queue. Such modification is useful during the debugging process of a virus in which the virus hide its real form; thus a novice virus code analyst is often unable to analyze such virus.
In modern processors all memory block have a set of bit that indicate the memory block attributes. Thus the memory block can be any combination of Read-Only, Writable, Executable or more attributes. Usually in modern computers the code area is marked as read-only. This can prevent viruses of modifying (or self-modifying) the code.
In such cases if the virus gains the ring0 privilege (in which it can modify anything it wants in the computer), it can change such memory attributes.
If not, a virus can move the code to a writable area of memory such as the stack or the heap, and jump into that area continuing the execution from that writable memory block.

3.2 Operating System Dependency

Traditionally, operating systems were hard-coded to a particular CPU architecture. Microsoft's first operating system (MS-DOS) supported Intel processors only. In '90s the need to support more CPU architectures with the same operating system was increasing. The operating systems began to be written in higher-level language, such as C/C++, and can be easy ported to other CPU architectures. Thus all UNIX derived systems (such as Linux) can run on different CPUs architectures. Even Microsoft's O.S. Windows NT was designed to support multiple CPU architectures.
Most computer viruses can operate only on a single operating system. They strictly depends on the environment they was designed for. It is feasible, but very rare, to create cross-environment viruses such as the *PeElf virus* that infects both Windows and Linux files. The virus programmer must be aware of the environment differences. For example the MS-DOS system calls can be invoked with the "int 0x21" instruction and by putting into the registers the appropriate numbers. In Linux system calls can be invoked with the "int 0x80" instruction and there is no correspondence with the registers values of the MS-DOS operating system. Even the functionality are different because the MS-DOS is an old mono-task mono-user system. Instead Linux in designed for newer CPU and can deal with memory block protection, task protection, and so on.
Microsoft's windows systems don't use the "int" instruction to call the system calls. This is done by calling the system APIs that are stored in various DLLs. (unofficially the programmer can call the "int 0x2E" which is totally undocumented by Microsoft). So, Linux, unlike windows, provides a direct way to interface with the kernel through the int 0x80 interface. Windows on the other hand, does not have a direct kernel interface. The system must be interfaced by loading the address of the function that needs to be executed from a DLL (Dynamic Link Library). The key difference between the two is the fact that the address of the functions found in windows will vary from OS version to OS version while the int 0x80 syscall numbers will remain constant. Windows programmers did this so that they could make any change needed to the kernel without any hassle; Linux on the contrary has fixed numbering system for all kernel level functions.
Some operating systems such as Microsoft's are backward compatible so a virus written for earlier O.S. like MS-DOS can replicate on Windows O.S. but can often fail. This because in newer operating systems some of the functionality or tricks are hided by the system or no more available. For example in MS-DOS we can use directly the I/O ports, but under Windows systems we cannot because they are handled by the system's virtual drivers and may be manipulated indirectly via the APIs.
A 32-bit Windows virus will infect only portable executable (PE) files and will not be able to replicate itself on DOS file format such as MZ. However, so-called multipartite viruses are able to infect several file formats or system areas, enabling them to jump from one operating environment to an other. The most important environmental dependency of a binary computer virus is the operating system itself.

3.3 Operating System Version Dependency

Some computer viruses depends not only on a particular operating system, but also on an actual system version. Young virus researchers often struggle to analyze such a virus. After a few minutes of unsuccessful test infections on their research system they might believe that a particular virus does not work at all. For example, the W95/Boza virus does not work on non-English releases of Windows 95, such as the Hungarian release of the operating system.
We have seen on the previous section (3.2) that different release version of Window may have different address for the system APIs. The programmer can interface with the kernel only with these APIs. But a virus programmer cannot compile its virus for every infection, so he must create a code that is able to interface with the kernel and get the APIs addresses on its own.
On Windows operative systems, there are multitudes of ways to find the addresses of the functions that the virus needs to use. There are two methods for addressing functions; you can find the desired function at runtime or use hard coded addresses.

3.3.1 runtime method

When a program is loaded into memory for execution, the memory map for all program is made of(every program in modern CPU architecture have a protected space of 4GB flat memory): the user space, the shared memory, the kernel, and the device drivers.

	begin memory	end memory
Application code and data	0x00000000	0x3FFFFFFF
Shared Memory	0x40000000	0x7FFFFFFF
Kernel	0x80000000	0xBFFFFFFF
Device Drivers	0xC0000000	0xFFFFFFFF

Table 3.1: Memory layout on program's execution

So the Kernel32.dll is loaded into the program's memory space. When we run an application, the code is invoked from a piece of code of the Kernel32, it's like the kernel makes a "CALL" instruction to the application. So, when the program is launched, it can retrieve the return address from the stack (the %esp register). The %eax register (in Figure 3.1) has a value similar to 0xBFF8XXXX, where XXXX has no importance. Infact the Win32 platform usually rounds up addresses to page alignment. Thus we can search for the Kernel32 PE heather at the beginning of all pages and retrieve the Kernell32 Base Address. This DLL holds LoadLibrary and GetProcAddress, the two functions needed to obtain any functions address.
Once we have the Kernell32's base address we can retrieve the full address of its func-

```
.data ;necessary for the TASM32/TLINK32
   db ?
.code
start:
   mov (%esp),%eax
   and $0xFFFF0000,%eax
   ret
end start
```

Figure 3.1: Retrieve the kernel's return address

tions by adding the offset we find in the Import or Export Table in the PE file heathers.

3.3.2 hard coded method

An other way for the injected code to call the system APIs, is to hard code at least the LoadLibrary and GetProcAddress address. We can use the program listed in Figure 3.2 for this purpose.

3.4 File System Dependency

Computer viruses also have file system dependencies. For most virus it does not matter whether the targeted files resides on a File Allocation Table (FAT) originally designed only for DOS; or the New Technology File System (NTFS) use by windows NT and newer Windows systems; or EXT2/EXT3 filesystem the default Linux FS; or a network file system. For such viruses as long as they are compatible with the operating environment's high-level file system interface, they work. They simply infect the file or store new files on the disk without paying attention to the actual storage format. However, other kinds of viruses depend strongly on the actual file system.
We have a look to how viruses can infect common file systems.

3.4.1 Cluster Viruses in FAT file system

The Bulgarian virus *DIR-II* spread itself by manipulating key structures of FAT-based file systems. The virus overwrites the pointer in the directory entry that points to the first cluster of a file with a value that directs the disk-read to the virus body. The virus stores the pointer to the real first cluster of each host program in an encrypted form in an unused part of the directory entry structure. This is used later to execute the real host from the disk after the virus has been loaded in memory.
Another common techniques is to place the virus code at the end of the file in the unused free space between the EOF (End of File) and the last sector of the last

```
#include <windows.h>
#include <stdio.h>
int main(int argc, char** argv)
{
   HMODULE hmod_libname;
   FARPROC fprc_func;

   printf("arwin - win32 address resolution program - by steve hanna - v.01\n");
   if(argc < 3)
   {
      printf("%s <Library Name> <Function Name>\n",argv[0]);
      exit(-1);
   }

   hmod_libname = LoadLibrary(argv[1]);
   if(hmod_libname == NULL)
   {
      printf("Error: could not load library!\n");
      exit(-1);
   }
   fprc_func = GetProcAddress(hmod_libname,argv[2]);

   if(fprc_func == NULL)
   {
      printf("Error: could find the function in the library!\n");
      exit(-1);
   }
   printf("%s is located at 0x%08x in %s\n",argv[2],(unsigned int)fprc_func,argv[1]);
}
```

Figure 3.2: Program that retrieves LoadLibrary and GetProcAddress adresses.

```
$ echo "test file">testfile.txt
$ cat testfile.txt
test file
$ cat testfile.txt/mystream
cat: testfile.txt/mystream: Not a directory
$ cat testfile.txt/rsrc
$ ll testfile.txt
-rw-r--r--   1 drugo  drugo  10 Jul 23 14:53 testfile.txt
$ echo "my stream text">testfile.txt/mystream
-bash: testfile.txt/mystream: Not a directory
$ ls -al testfile.txt/rsrc
-rw-r--r--   1 drugo  drugo  0 Jul 23 14:53 testfile.txt/rsrc
$ echo "my test string">testfile.txt/rsrc
$ ls -al testfile.txt/rsrc
-rw-r--r--   1 drugo  drugo  15 Jul 23 15:00 testfile.txt/rsrc
$ cat testfile.txt/rsrc
my test string
$
```

Figure 3.3: The resource fork on HFS (Mac OS X) file system

cluster. Infact the disk is divided into sectors (usually of 512 bytes each), and sectors are grouped into clusters. So it is rare that a program fills without leaving unused sectors (or totally used sectors) of the last cluster assigned to it from the operating system. This is a good place to store the virus code because it wont increase the size of the host program on the disk.

3.4.2 HFS File System

A little-known feature of the HFS file system (used actually on Mac OS X 10.4) is the so called *Resource fork*. The resource fork is an invisible file stream (named **/rsrc**) associated to a regular file so that it is easier to store a variety of additional information, such as allowing the system to display the correct icon for a file and open it without the need for a file extension in the file name. This file stream is completely invisible to the user when it performs a list of the files in the directory but smart users can notice differences between the actual file size before and after have written data into the resource fork(Figure 3.3). A virus writer can use this feature to implement a *Companion virus* (Appendix B).

3.4.3 NTFS File System

The NTFS file system was introduced with Windows NT operating system. Windows NT had to support multiple-fork files because the server version was intended to

```
c:\> echo this is a test > stream.dat:text
c:\> more <stream.dat:text
this is a test
```

Figure 3.4: NTFS multiple streams.

service Macintosh computers. So on NTFS, a file can contain multiple streams on the disk. The "main stream" is the actual file itself. For instance, notepad.exe's code can be found in the main stream of the file. A file can contain not only one additional stream as it was on Apple's HFS (3.4.2), but a variety of alternative streams (with customizable names) can be created and associated to a file (Figure 3.4). This is a bigger security problem that HFS file system, because in HFS the alternative stream has always the same constant name, so it is simple to check if a file has been infected, but in NTFS we cannot find it easily.

3.5 File Format Dependency

Viruses can be classified according to the file object they can infect. This short section is an introduction to common binary format infectors.

3.5.1 COM Viruses on DOS

A COM file format is a flat format that doesn't have any structure needed by the system loader to map it into memory. The DOS simply makes a copy of the file at a memory location that starts 256 bytes after the beginning of the chosen segment. The first 256 byte are reserved by the operating system for the **Program Segment Prefix** (PSP for short). The PSP holds the state of the running program seen by the operative system (Figure 3.5). So the entry point is fixed (CS:100h). A common technique for infecting such kind of files is to place a "jump" instruction at the beginning of the file that points to the virus code usually appended to the end of the file. Once the virus has finished it can rejoin the original execution flow.
The infection strategies are explained more in detail in the next chapter.

3.5.2 EXE Viruses on DOS

The EXE file format was introduced by Microsoft to break the 64K limit that affected the COM files. The file is seen now as a group of segment each of 64K maximum.
The EXE file format has a little structure (Figure 3.6) used by the system loader. The entry-point is independent by the PSP.
A common technique for infecting EXE files is to modify the program's entry point from the EXE structure at offset 0x14. This will point to the virus code injected,

```
Offset Size       Description

00   word      machine code INT 20 instruction (CDh 20h)
02   word      top of memory in segment (paragraph) form
04   byte      reserved for DOS, usually 0
05  5bytes     machine code instruction long call to the DOS
               function dispatcher (obsolete CP/M)
06   word      .COM programs bytes available in segment (CP/M)
0A   dword      INT 22 terminate address;  DOS loader jumps to this
                address upon exit;  the EXEC function forces a child
                process to return to the parent by setting this
                vector to code within the parent (IP,CS)
0E   dword      INT 23 Ctrl-Break exit address; the original INT 23
                vector is NOT restored from this pointer (IP,CS)
12   dword      INT 24 critical error exit address; the original
                INT 24 vector is NOT restored from this field (IP,CS)
16   word       parent process segment addr (Undoc. DOS 2.x+)
                COMMAND.COM has a parent id of zero, or its own PSP
18  20bytes     file handle array (Undocumented DOS 2.x+); if handle
                array element is FF then handle is available.  Network
                redirectors often indicate remotes files by setting
                these to values between 80-FE.
2C   word       segment address of the environment, or zero (DOS 2.x+)
2E   dword      SS:SP on entry to last INT 21 function (Undoc. 2.x+) ∅
32   word       handle array size (Undocumented DOS 3.x+)
34   dword      handle array pointer (Undocumented DOS 3.x+)
38   dword      pointer to previous PSP (deflt FFFF:FFFF, Undoc 3.x+) ∅
3C  20bytes     unused in DOS before 4.01  ∅
50   3bytes     DOS function dispatcher CDh 21h CBh (Undoc. 3.x+) ∅
53   9bytes     unused
5C  36bytes     default unopened FCB (File Control Block) #1 (parts overlayed by FCB #2)
6C  20bytes     default unopened FCB #2 (overlays part of FCB #1)
80   byte       count of characters in command tail;  all bytes
                following command name;  also default DTA (Disk allocation Table) (128 b
81 127bytes     all characters entered after the program name followed
                by a CR byte


- offset 5 contains a jump address which is 2 bytes too low for
  PSP's created by the DOS EXEC function in DOS 2.x+  ∅
- program name and complete path can be found after the environment
  in DOS versions after 3.0.  See offset 2Ch.
```

Figure 3.5: The Program Segment Prefix (PSP) structure.

```
 Offset   Size   Description
00      word    "MZ" - Link file .EXE signature
                Mark Zbikowski: Microsoft's engineer who created the EXE format.
02      word    length of image mod 512
04      word    size of file in 512 byte pages
06      word    number of relocation items following header
08      word    size of header in 16 byte paragraphs, used to locate
                the beginning of the load module
0A      word    min # of paragraphs needed to run program
0C      word    max # of paragraphs the program would like
0E      word    offset in load module of stack segment (in paragraphs)
10      word    initial SP value to be loaded
12      word    negative checksum of pgm used while by EXEC loads pgm
14      word    program entry point, (initial IP value and CS file offset)
16      word    offset in load module of the code segment (in paragraphs)
18      word    offset in .EXE file of first relocation item
1A      word    overlay number (0 for root program)

    * relocation table and the program load module follow the header
    * relocation entries are 32 bit values representing the offset into
      the load module needing patched
    * once the relocatable item is found, the CS register is added to the
      value found at the calculated offset

Registers at load time of the EXE file are as follows:
AX      contains number of characters in command tail, or 0
BX:CX   32 bit value indicating the load module memory size DX zero
SS:SP   set to stack segment if defined else, SS = CS and SP=FFFFh or top of memory.
DS      set to segment address of EXE header
ES      set to segment address of EXE header
CS:IP   far address of program entry point, (label on "END" statement of program)
```

Figure 3.6: The EXE format structure.

and the original entry point is saved by the virus and used later to return to original execution flow.

3.5.3 PE (Portable Executable) Viruses on Windows

The PE file format for Windows NT introduces a completely new structure to developers familiar with the Windows and MS-DOS environments. Yet developers familiar with the UNIX environment will find that the PE file format is similar to, if not based on, the COFF specification.
The entire format consists of an MS-DOS MZ header, followed by a real-mode stub program, the PE file signature, the PE file header, the PE optional header, all of the section headers, and finally, all of the section bodies (Figure 3.7).
We now have a look the the most important fields of the file structure and we explain the common infection techniques (see ??) later.

MS-DOS/Real-Mode Header

The MS-DOS header is not new for the PE file format. It is the same MS-DOS header that has been around since version 2 of the MS-DOS operating system. The main reason for keeping the same structure intact at the beginning of the PE file format is so that, when you attempt to load a file created under Windows version 3.1 or earlier, or MS DOS version 2.0 or later, the operating system can read the file and understand that it is not compatible. In other words, when you attempt to run a Windows NT executable on MS-DOS version 6.0, you get this message: "This program cannot be run in DOS mode." If the MS-DOS header was not included as the first part of the PE file format, the operating system would simply fail the attempt to load the file and offer something completely useless, such as: "The name specified is not recognized as an internal or external command, operable program or batch file.".
The MS-DOS header occupies the first 64 bytes of the PE file. A structure representing its content is described below:

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
   USHORT e_magic;         // Magic number
   USHORT e_cblp;          // Bytes on last page of file
   USHORT e_cp;            // Pages in file
   USHORT e_crlc;          // Relocations
   USHORT e_cparhdr;       // Size of header in paragraphs
   USHORT e_minalloc;      // Minimum extra paragraphs needed
   USHORT e_maxalloc;      // Maximum extra paragraphs needed
   USHORT e_ss;            // Initial (relative) SS value
   USHORT e_sp;            // Initial SP value
   USHORT e_csum;          // Checksum
```

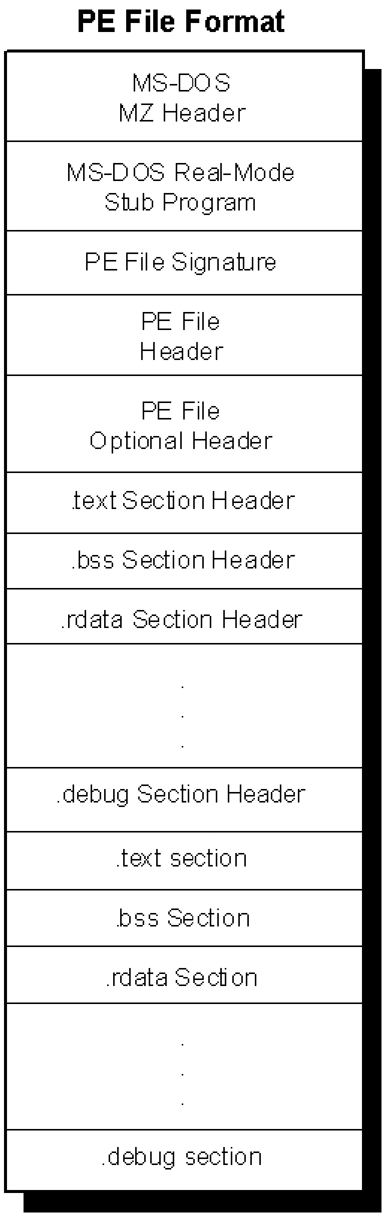


Figure 3.7: The PE File Format Structure.

```
    USHORT e_ip;            // Initial IP value
    USHORT e_cs;            // Initial (relative) CS value
    USHORT e_lfarlc;        // File address of relocation table
    USHORT e_ovno;          // Overlay number
    USHORT e_res[4];        // Reserved words
    USHORT e_oemid;         // OEM identifier (for e_oeminfo)
    USHORT e_oeminfo;       // OEM information; e_oemid specific
    USHORT e_res2[10];      // Reserved words
    LONG   e_lfanew;        // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

The first field, e_magic, is the so-called magic number. This field is used to identify an MS-DOS-compatible file type. All MS-DOS-compatible executable files set this value to 0x54AD, which represents the ASCII characters MZ. MS-DOS headers are sometimes referred to as MZ headers for this reason. Many other fields are important to MS-DOS operating systems, but for Windows NT, there is really one more important field in this structure. The final field, e_lfanew, is a 4-byte offset into the file where the PE file header is located. It is necessary to use this offset to locate the PE header in the file. For PE files in Windows NT, the PE file header occurs soon after the MS-DOS header with only the real-mode stub program between them.

PE File Header and Signature

All possible Signature accepted at the moment from the loader are listed below (from winnt.h):

```
#define IMAGE_DOS_SIGNATURE             0x5A4D      // MZ
#define IMAGE_OS2_SIGNATURE             0x454E      // NE
#define IMAGE_OS2_SIGNATURE_LE          0x454C      // LE
#define IMAGE_NT_SIGNATURE              0x00004550  // PE00
```

The PE header is a 0x20-byte data structure describing the fundamental file characteristics and containing the PE/x0/x0 signature, which is used ti identify the file as PE.

```
typedef struct _IMAGE_FILE_HEADER {
    USHORT  Machine;
    USHORT  NumberOfSections;
    ULONG   TimeDateStamp;
    ULONG   PointerToSymbolTable;
    ULONG   NumberOfSymbols;
    USHORT  SizeOfOptionalHeader;
    USHORT  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;

#define IMAGE_SIZEOF_FILE_HEADER            20
```

The **Machine** field indicates the type of CPU for which the file has been compiled. The file won't load on I386 machines if this field contains anything other than 0x14C.
The **NumberOfSection** field depends on the specific features of the way the loader is implemented or depends on the way the compiler generate the code. This field can be modified by virus if it performs an injection of code by adding a section.
The section table can be located by the sum of the e_lfanew plus the size of the image file header and the size of the optional header.

PE Optional Header

The optional header contains most of the meaningful information about the executable image, such as initial stack size, program entry point location, preferred base address, operating system version, section alignment information, and so forth. The term "optional" is not an appropriate choice for this header. This header is mandatory, is not optional. This can only be understood in relation to the fact that, when the PE was under construction, the situation was different, and this term is only a legacy.
The IMAGE_OPTIONAL_HEADER structure represents the optional header as follows:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //
    USHORT  Magic;
    UCHAR   MajorLinkerVersion;
    UCHAR   MinorLinkerVersion;
    ULONG   SizeOfCode;
    ULONG   SizeOfInitializedData;
    ULONG   SizeOfUninitializedData;
    ULONG   AddressOfEntryPoint;
    ULONG   BaseOfCode;
    ULONG   BaseOfData;
    //
    // NT additional fields.
    //
    ULONG   ImageBase;
    ULONG   SectionAlignment;
    ULONG   FileAlignment;
    USHORT  MajorOperatingSystemVersion;
    USHORT  MinorOperatingSystemVersion;
    USHORT  MajorImageVersion;
    USHORT  MinorImageVersion;
    USHORT  MajorSubsystemVersion;
    USHORT  MinorSubsystemVersion;
    ULONG   Reserved1;
    ULONG   SizeOfImage;
    ULONG   SizeOfHeaders;
    ULONG   CheckSum;
    USHORT  Subsystem;
    USHORT  DllCharacteristics;
    ULONG   SizeOfStackReserve;
    ULONG   SizeOfStackCommit;
    ULONG   SizeOfHeapReserve;
    ULONG   SizeOfHeapCommit;
    ULONG   LoaderFlags;
    ULONG   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

where:

Magic: This is the state of the mapped file. If this contains anything other than 0x10 (the signature of the executable image), the file will fail to load. PE64 files bear the 0x20 signature, because all addresses are 64-bit.

SizeOfCode: This filed describes the rounded-up size of all executable sections. Usually viruses do not fix the value when adding a new code section to the host program.

AddessOfEntryPoint: The address where the execution of the image begins. This values is an RVA (Relative Virtual Address) that normally points to the .text (or CODE) section. RVA is relative to disk file image. The equivalent memory address is: $MEMORY = ImageBase + RVA$.
This field can be modified by the virus to point to its viral code. If the virus adds a new section and points the address of entry point to this new section, many antiviral product can notice that the .text section is no more the main entry point and can label the file as infected.

ImageBase: When the linker creates a PE executable, it assumes that the image will be mapped to a specific memory location. That address is stored in this field. That address is a preferred address (currently 0x400000). If the table of relocatable elements is present, the file can be loaded by any address other than that specified in the header. This field is used by most viruses before infection to calculate the actual address of certain items, but is not usually changed.

SectionAlignment: When the executable is mapped into memory, each section must start at a virtual address that is a multiple of this value. This field minimum is 0x1000 (4096 bytes). Most Win32 viruses use this field to calculate the correct location for the virus body but do not change the field value.

FileAlignment: In the PE file, the raw data starts at a multiple of this location. Viruses do not change this values but use it in a similar way to SectionAlignment.

SizeOfImage: When the linker creates the image, it calculates the total size of the portion of the image that the loader has to load. This includes the size of the region starting at the image base up through the end of the last section. The end of the last section is rounded-up to the nearest multiple of the SectionAlignment. Almost every PE infection method uses and changes the SizeOfImage value of the PE header. Many virus calculate this field incorrectly and the loader as a result won't load the infected file.

Checksum: This is a checksum of the file. Most executables contain zero in this field. All DLLs and drivers, however, must have a correct checksum. Windows 95 loader simply ignores the checking of this field before loading DLLs. This field is used by some viruses to represent an infection marker to avoid double infections. Another set of viruses recalculates it to hide an infection even better.

Data Directories

The data directory indicates where to find other important components of executable information in the file. It is really nothing more than an array of IMAGE_DATA_DIRECTORY structures that are located at the end of the optional header structure. The current PE file format defines 16 possible data directories, 11 of which are now being used.

```
// Directory Entries

// Export Directory
#define IMAGE_DIRECTORY_ENTRY_EXPORT         0
// Import Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT         1
// Resource Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE       2
// Exception Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION      3
// Security Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY       4
// Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_BASERELOC      5
// Debug Directory
#define IMAGE_DIRECTORY_ENTRY_DEBUG          6
// Description String
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT      7
// Machine Value (MIPS GP)
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR      8
// TLS Directory
#define IMAGE_DIRECTORY_ENTRY_TLS            9
// Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10
```

Each data directory is basically a structure defined as an IMAGE_DATA_DIRECTORY. And although data directory entries themselves are the same, each specific directory type is entirely unique.

```
 typedef struct _IMAGE_DATA_DIRECTORY {
    ULONG   VirtualAddress;
    ULONG   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Each data directory entry specifies the size and relative virtual address of the directory. To locate a particular directory, you determine the relative address from the data directory array in the optional header. Then use the virtual address to determine which section the directory is in. Once you determine which section contains the directory, the section header for that section is then used to find the exact file offset location of the data directory.
So to get a data directory, you first need to know about sections, which are described next.
Some times the virus has to follow some of the addresses into the data directories and change or use the values according to its needs. The two most important data directories are "entry export" and the "entry import". The first is is a pointer to the table of exported functions and data and will be encountered mainly in DLLs and drivers. The second is a pointer to the table of imported functions (APIs) used for communicating with the outside world.

PE File Section

The PE file specification consists of the headers defined so far and a generic object called a section. Sections contain the content of the file, including code, data, resources, and other executable information. Each section has a header and a body (the raw data). Section headers are described below, but section bodies lack a rigid file structure. They can be organized in almost any way a linker wishes to organize them, as long as the header is filled with enough information to be able to decipher the data.
Section headers are located sequentially right after the optional header in the PE file format. Each section header is 40 bytes with no padding between them. Section headers are defined as in the following structure:

```
#define IMAGE_SIZEOF_SHORT_NAME             8

typedef struct _IMAGE_SECTION_HEADER {
    UCHAR   Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            ULONG   PhysicalAddress;
            ULONG   VirtualSize;
    } Misc;
    ULONG   VirtualAddress;
    ULONG   SizeOfRawData;
    ULONG   PointerToRawData;
    ULONG   PointerToRelocations;
    ULONG   PointerToLinenumbers;
    USHORT  NumberOfRelocations;
    USHORT  NumberOfLinenumbers;
    ULONG   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

where:

Name:	Each section header has a name field up to eight characters long, for which the first character must be a period.
PhysicalAddress:	RVA address. VirtualSize is a union field that is not currently used.
VirtualAddress:	This field identifies the virtual address in the process's address space to which to load the section. The actual address is created by taking the value of this field and adding it to the ImageBase virtual address in the optional header structure. Keep in mind, though, that if this image file represents a DLL, there is no guarantee that the DLL will be loaded to the ImageBase location requested. So once the file is loaded into a process, the actual ImageBase value should be verified programmatically using GetModuleHandle.
SizeOfRawData:	This field indicates the FileAlignment-relative size of the section body. The actual size of the section body will be less than or equal to a multiple of FileAlignment in the file. Once the image is loaded into a process's address space, the size of the section body becomes less than or equal to a multiple of SectionAlignment
PointerToRawData:	This is an offset to the location of the section body in the file.
Characteristics:	Defines the section characteristics (attributes).

PointerToRelocations, PointerToLinenumbers, NumberOfRelocations, NumberOfLinenumbers. None of these fields are used in the PE file format.

To infect a PE file, the virus has to manipulate correctly the file structures according to the infection technique used. We will discuss later those infection techniques (see XX).

3.5.4 ELF Viruses on UNIX

The executable and linking format (ELF) was originally developed by Unix System Laboratories and is rapidly becoming the standard in file formats. The ELF standard

is growing in popularity because it has greater power and flexibility than the a.out and COFF binary formats. ELF now appears as the default binary format on operating systems such as Linux, Solaris 2.x, and SVR4. Some of the capabilities of ELF are dynamic linking, dynamic loading, imposing runtime control on a program, and an improved method for creating shared libraries. The ELF representation of control data in an object file is platform independent, an additional improvement over previous binary formats. The ELF representation permits object files to be identified, parsed, and interpreted similarly, making the ELF object files compatible across multiple platforms and architectures of different size.
The three main types of ELF files are executable, relocatable, and shared object files. These file types hold the code, data, and information about the program that the operating system and/or link editor need to perform the appropriate actions on these files. The three types of files are summarized as follows:

- An executable file supplies information necessary for the operating system to create a process image suitable for executing the code and accessing the data contained within the file.
- A relocatable file describes how it should be linked with other object files to create an executable file or shared library.
- A shared object file contains information needed in both static and dynamic linking.

There are two views for each of the three file types described in the previous section. These views support both the linking and execution of a program. The two views are summarized in Figure 3.8 where the view on the left of the figure is the link view and the view on the right of the figure is the execution view. The link view of the ELF object file is partitioned by sections and the execution view of the ELF object file is partitioned by segments. Thus, the programmer interested in obtaining section information about the program items such as symbol tables, relocation, specific executable code or dynamic linking information will use the link view; the programmer interested in obtaining segment information such as the location of the text segment or data segment will use the execution view. The ELF access library, libelf, provides a programmer with tools to extract and manipulate ELF object file contents for either view. The ELF header describes the layout of the rest of the object file. It provides information on where and how to access the other sections. The Section Header Table gives the location and description of the sections and is mostly used in linking. The Program Header Table provides the location and description of segments and is mostly used in creating a programs' process image. Both sections and segments hold the majority of data in an object file including: instructions, data, symbol table, relocation information, and dynamic linking information.
The ELF header structure is :

```
#define EI_NIDENT       16
```

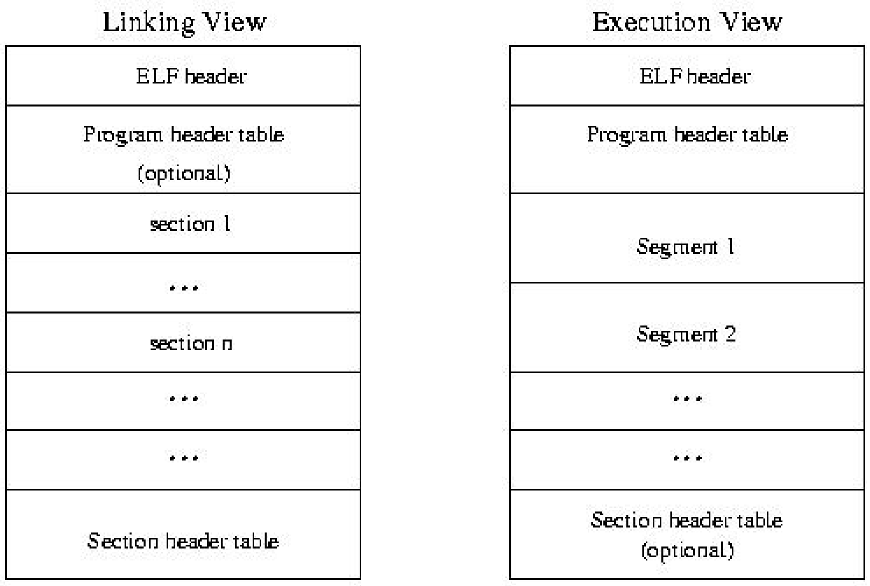


Figure 3.8: Linking and Execution Views.

```
typedef struct {
    unsigned char       e_ident[EI_NIDENT];
    Elf32_Half          e_type;
    Elf32_Half          e_machine;
    Elf32_Word          e_version;
    Elf32_Addr          e_entry;
    Elf32_Off           e_phoff;
    Elf32_Off           e_shoff;
    Elf32_Word          e_flags;
    Elf32_Half          e_ehsize;
    Elf32_Half          e_phentsize;
    Elf32_Half          e_phnum;
    Elf32_Half          e_shentsize;
    Elf32_Half          e_shnum;
    Elf32_Half          e_shstrndx;
} Elf32_Ehdr;
```

The meaning of the fields are listed below:

e_ident: The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents. Complete descriptions appear below, in "ELF Identification."

e_type: This member identifies the object file type.

Name	Value	Meaning
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_LOPROC	0xff00	Processor-specific
ET_HIPROC	0xffff	Processor-specific

e_machine: This member's value specifies the required architecture for an individual file.

Name	Value	Meaning
EM_NONE	0	No machine
EM_M32	1	AT&T WE 32100
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000

Other values are reserved and will be assigned to new machines as necessary. Processor-specific ELF names use the machine name to distinguish them. For example, the flags mentioned below use the prefix EF_; a flag named WIDGET for the EM_XYZ machine would be called EF_XYZ_WIDGET.

e_version: This member identifies the object file version.

Name	Value	Meaning
EV_NONE	0	Invalid version
EV_CURRENT	1	Current version

The value 1 signifies the original file format; extensions will create new versions with higher numbers. The value of EV_CURRENT, though given as 1 above, will change as necessary to reflect the current version number.

e_entry: This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

e_phoff: This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.

e_shoff: This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.

e_flags: This member holds processor-specific flags associated with the file. Flag names take the form EF_<machine>_<flag>.

e_ehsize: This member holds the ELF header's size in bytes.

e_phentsize: This member holds the size in bytes of one entry in the file's program header table; all entries are the same size.

e_phnum: This member holds the number of entries in the program header table. Thus the product of e_phentsize and e_phnum gives the table's size in bytes. If a file has no program header table, e_phnum holds the value zero.

e_shentsize: This member holds a section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.

e_shnum: This member holds the number of entries in the section header table. Thus the product of e_shentsize and e_shnum gives the section header table's size in bytes. If a file has no section header table, e_shnum holds the value zero.

e_shstrndx: This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value SHN_UNDEF.

The program header gives a runtime-views of the file.

```
typedef struct {
    Elf32_Word p_type;
    Elf32_Off p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
} Elf32_Phdr;
```

where:

p_type: This member tells what kind of segment this array element describes or how to interpret the array element's information. Type values and their meanings appear below.

p_offset: This member gives the offset from the beginning of the file at which the first byte of the segment resides.

p_vaddr: This member gives the virtual address at which the first byte of the segment resides in memory.

p_paddr: On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Because System V ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.

p_filesz: This member gives the number of bytes in the file image of the segment; it may be zero.

p_memsz: This member gives the number of bytes in the memory image of the segment; it may be zero.

p_flags: This member gives flags relevant to the segment. Defined flag values appear below.

p_align: Loadable process segments must have congruent values for p_vaddr and p_offset, modulo the page size. This member gives the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, p_align should be a positive, integral power of 2, and p_vaddr should equal p_offset, modulo p_align. Possible values of this fields can be found in elf.h file.

An executable or shared object file's base address is calculated during execution from three values: the memory load address, the maximum page size, and the lowest virtual address of a program's loadable segment. The virtual addresses in the program headers might not represent the actual virtual addresses of the program's memory image. To compute the base address, one determines the memory address associated with the lowest p_vaddr value for a PT_LOAD segment. One then obtains the base address by truncating the memory address to the nearest multiple of the maximum page size. Depending on the kind of file being loaded into memory, the memory address might or might not match the p_vaddr values.
The section header describes the sections of the ELF file.

```
typedef struct {
      Elf32_Word sh_name;
      Elf32_Word sh_type;
      Elf32_Word sh_flags;
      Elf32_Addr sh_addr;
      Elf32_Off sh_offset;
      Elf32_Word sh_size;
      Elf32_Word sh_link;
```

```
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
} Elf32_Shdr;
```

where:

sh_name: This member specifies the name of the section. Its value is an index into the section header string table section, giving the location of a null-terminated string.

sh_type: This member categorizes the section's contents and semantics. Section types and their descriptions appear below.

sh_flags: Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions appear below.

sh_addr: If the section will appear in the memory image of a process, this member gives the address at which the section's first byte should reside. Otherwise, the member contains 0.

sh_offset: This member's value gives the byte offset from the beginning of the file to the first byte in the section. One section type, SHT_NOBITS described below, occupies no space in the file, and its sh_offset member locates the conceptual placement in the file.

sh_size: This member gives the section's size in bytes. Unless the section type is SHT_NOBITS, the section occupies sh_size bytes in the file. A section of type SHT_NOBITS may have a non-zero size, but it occupies no space in the file.

sh_link: This member holds a section header table index link, whose interpretation depends on the section type. A table below describes the values.

sh_info: This member holds extra information, whose interpretation depends on the section type. A table below describes the values.

sh_addralign: Some sections have address alignment constraints. For example, if a section holds a doubleword, the system must ensure doubleword alignment for the entire section. That is, the value of sh_addr must be congruent to 0, modulo the value of sh_addralign. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

sh_entsize: Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, this member gives the size in bytes of each entry. The member contains 0 if the section does not hold a table of fixed-size entries.

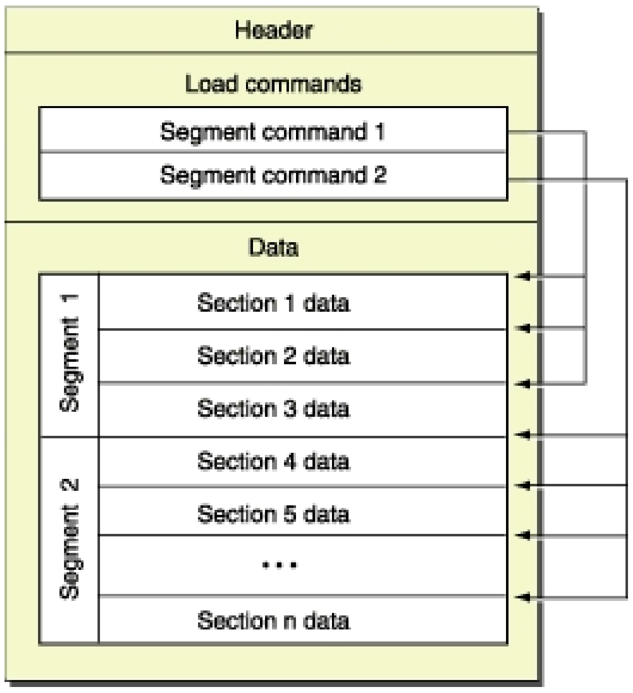


Figure 3.9: The Mach-O File Format

3.5.5 Mach-O Viruses on Mac OS X

The Mach-O file format belongs to the modern files structure as PE and ELF. It is composed by a main header containing information common to all files, and a variable number of segments (Load Commands) which can contain zero or more sections (Figure 3.9).
The Mach header appears at the beginning of the object file. Only information that's truly general to the file is contained in the Mach header. Other information is put in the load commands that follow.
The format of the Mach header is:

```
struct mach_header {
   unsigned long  magic;      /* Mach magic number identifier */
   cpu_type_t     cputype;    /* cpu specifier */
   cpu_subtype_t  cpusubtype; /* machine specifier */
   unsigned long  filetype;   /* type of file */
   unsigned long  ncmds;      /* number of load commands */
   unsigned long  sizeofcmds; /* size of all load commands */
   unsigned long  flags;      /* flags */
};
```

The load commands appear directly after the Mach header. They are variable in size. The number of load commands and the total size of the commands are given in the

ncmds and sizeofcmds fields of the mach_header structure.
All load commands must have as their first two fields cmd and cmdsize. The following structure is the minimum form of a load command:

```
struct load_command {
   unsigned long  cmd;      /* type of load command */
   unsigned long  cmdsize;  /* total size of command in bytes */
};
```

Constants for the cmd field of the load_command structure are:

```
#define LC_SEGMENT     0x1   /* file segment to be mapped */
#define LC_SYMTAB      0x2   /* link-edit stab symbol table info
                                (obsolete) */
#define LC_SYMSEG      0x3   /* link-edit gdb symbol table info */
#define LC_THREAD      0x4   /* thread */
#define LC_UNIXTHREAD  0x5   /* UNIX thread (includes a stack) */
#define LC_LOADFVMLIB  0x6   /* load a fixed VM shared library */
#define LC_IDFVMLIB    0x7   /* fixed VM shared library id */
#define LC_IDENT       0x8   /* object identification information
                                (obsolete) */
#define LC_FVMFILE      0x9   /* fixed VM file inclusion */
```

We give below the explanation of two of the most significant Load Commands.

The LC_SEGMENT Load Command

The LC_SEGMENT load command indicates that a part of this file is to be mapped into the task's address space. The size of this segment in memory, vmsize, can be equal to or larger than the amount to map from this file, filesize. The file, starting at fileoff, is mapped to the beginning of the segment in memory at vmaddr. The rest of the memory of the segment, if any, is allocated zero-fill on demand.

```
struct segment_command {
   unsigned long  cmd;          /* LC_SEGMENT */
   unsigned long  cmdsize;      /* includes size of section
                                   structures */
   char           segname[16];  /* segment's name */
   unsigned long  vmaddr;       /* segment's memory address */
   unsigned long  vmsize;       /* segment's memory size */
   unsigned long  fileoff;      /* segment's file offset */
   unsigned long  filesize;     /* amount to map from file */
   vm_prot_t      maxprot;      /* maximum VM protection */
   vm_prot_t      initprot;     /* initial VM protection */
   unsigned long  nsects;       /* number of sections */
   unsigned long  flags;        /* flags */
};
```

The segment's maximum virtual memory protection and initial virtual memory protection are specified by the maxprot and initprot fields. The values for these fields are set to some combination of the constants defined in the header file vm/vm_prot.h:

```
#define VM_PROT_NONE    ((vm_prot_t) 0x00)
#define VM_PROT_READ    ((vm_prot_t) 0x01)  /* read permission */
#define VM_PROT_WRITE   ((vm_prot_t) 0x02)  /* write permission */
#define VM_PROT_EXECUTE ((vm_prot_t) 0x04)  /* execute permission */

/* The default protection for newly created virtual memory */
#define VM_PROT_DEFAULT  \
(VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXECUTE)

/* Maximum privileges possible, for parameter checking. */
#define VM_PROT_ALL  \
(VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXECUTE)
```

A segment's address and virtual memory protection are set at link edit time.
A segment is made up of zero or more sections. If the segment contains sections, the section structures directly follow the segment command and their size is reflected in the cmdsize field.
If sections have the same section name and are going into the same segment, they're combined by the link editor. The resulting section is aligned to the maximum alignment of the combined sections and is the new section's alignment. The combined sections are aligned to their original alignment in the combined section. Any padded bytes used to get the specified alignment are zero-filled.

```
struct section {
char  sectname[16];       /* section's name */
char  segname[16];        /* segment the section is in */
unsigned long  addr;      /* section's memory address */
unsigned long  size;      /* section's size in bytes */
unsigned long  offset;    /* section's file offset */
unsigned long  align;     /* section's alignment */
unsigned long  reloff;    /* file offset of relocation entries */
unsigned long  nreloc;    /* number of relocation entries */
unsigned long  flags;     /* flags */
unsigned long  reserved1; /* reserved */
unsigned long  reserved2; /* reserved */
};
```

S_ZEROFILL is used for the uninitialized data sections; sections with literal flags cause the link editor to coalesce redundant literals into sections and perform the proper relocation, resulting in a smaller file.

The LC_THREAD and LC_UNIXTHREAD Load Commands

Thread commands contain machine-specific data structures suitable for use in the thread state primitives.The machine-specific data structures follow the struct thread_command or struct unixthread_command as follows: Each flavor of machine-specific data structure is preceded by an unsigned long constant for the flavor of that data structure and an unsigned long that's the count of longs of the size of the state data structure, and then the state data structure follows that. This triple may be repeated for many flavors.

```
struct thread_command {
   unsigned long  cmd;          /* LC_THREAD or LC_UNIXTHREAD */
   unsigned long  cmdsize;      /* sizeof(struct thread_command) */
   /* unsigned long  flavor        flavor of thread state */
   /* unsigned long  count         count of longs in thread state */
   /* struct XXX_thread_state state   flavor's thread state */
   /* . . . */
};
```

The LC_UNIXTHREAD load command describes the register values for the main thread in the file.Yes, that includes Program Counter register (EIP on intel's machine or SV1 on PowerPC architectures). By changing this value we can control the main entry point of the program.

3.6 Interpreted Environment Dependency

Interpreted environments are very common in modern computers environment. The interpreted scrips have the advantage that are platform-independent. Unix systems are full of bash and perl scripts. Modification of most script file is allowed. Most frequently, scripts comprise hundreds of code lines, in which it is easy to get lost. Capabilities of scripts are comparable to those of high-level programming languages such as C,Basic, Pascal.
Also almost every major applications support users with programmability. This is the case of Microsoft's Word and Excel, or even GNU applications such as Octave, gdb. So every programmable environment can lead to a possible virus.

Interpreted language	**environment**
Perl/Python	practically omnipresent in unix derived environments
Bash	unix
VBA/VBS and Macro	Visual Basic scripts can infect Word and Excel documents and the Windows OS
Jscript	can be dangerous if invoke an ActiveX communication objects
PHP	can run in command line mode.

Table 3.2: Some of the most known interpreted languages that can be used to write viruses.

Chapter 4

Classification of Infection strategies

4.1 Boot Viruses

Boot sector viruses take advantage of the boot process of personal computers (PCs). Because most computers do not contain an operating system in their read-only memory (ROM), they need to load the system from somewhere else, such as from a disk or from a network.
When the PC starts up, it gives control to a ROM-BIOS routine that has to load the boot sector from the selected disk. The boot sector is loaded into the 0:7C00 memory address and the control is given to it. Typically the boot sequence try to read a diskette. If no disk is present then it would read the hard disks boot record.
A removable disk structure has a disk boot sector (DBR Disk Boot Record) that contains all the information about the disk structure such as number of sectors, number of tracks, sectors per track and so on.
Indeed a hard disk is much bigger than a floppy and can be divided into a maximum of four partitions (each can be divided again). So in hard disks there is a Master Boot Record (MBR) that has a table of those partitions and a simple boot-strap code that let load the operative system in the selected partition. A little trick that some MBR viruses use is to change the CMOS settings of the BIOS so a PC think that it has no floppy drives. Thus, the PC will boot using the infected MBR and then, when the virus is executed, it checks if there is a floppy drive present.
We describe a few techniques below that can be applied to both MBR and DBR.

4.1.1 Replacing the Boot Record without saving it

This technique overwrites the boot strap code leaving the Partition Table (PT) entries in place in case of MBR without saving the boot sector anywhere. When infecting a Boot Sector the virus must perform the function of the original boot sector code.
When disinfecting the original bot sector cannot be retrieved any more. So a new one (standard) is placed into the right sector(s) of the disk.

4.1.2 Replacing the Boot Record making a copy of it

A class of boot viruses replaces the original boot sector by overwriting it and saving it in a free or unused sector of the disk. Rudimental viruses just store the original sector in a free sector with the hope that this sector is not used in the future.
Advanced viruses take care of the original bot sector safety because if the disk don't boot correctly the virus wont spread any more. There are several techniques to safe the original boot sector. The virus can mark the sector as a BAD sector so the operative system wont use this sector for storing information. Ora virus can resize a disk partition and store the original boot sector out of all the partitions on the disk. In that manner the sector is not used by the operative system.

4.1.3 Boot Viruses that mark sectors as BAD

Some times a virus is too small ti fit into a single sector (the default is 512 bytes). So multiple sectors are required to store the virus into the disk. To protect from being overwrited, the virus can mark the sectors used to store its own code (or a copy of the original boot sector) as a BAD sector. Thus these sectors wont be used by the operative system even if they are not physically damaged.

4.2 File Infection Techniques

4.2.1 Overwriting Viruses

Some viruses simply locate another file on the disk and overwrite it with their own copy. Of course this is a very primitive technique, but it is certainly the easiest approach of all.
Overwriting viruses cannot be disinfected from a system. Infected files must be deleted from the disk and restored from backups.
Overwriting viruses are easily discovered from the users because programs don't act as they should any more. So this class of viruses don't spread very much.
Anyway during the early 1990s, many virus writers attempted to write the shortest possible binary virus using this technique. Some of the virus are as short as 22 bytes (*Trivial.22*).
The algorithm for such viruses is simple:

- Search for any (*.*) new host file in the current directory.
- open file for writing.
- write the virus code on top of the host.

The shortest viruses are often unable to infect more than a single host program in the same directory in which the virus was executed. This because finding the next host file would be "as expensive" as a couple of bytes of extra code.

```
+----------------+
|PPPPPPPPPPPPPPPP|    clean program
+----------------+

+----+-----------+
|VVVV|PPPPPPPPPPP|    infected program
+----+-----------+
```

4.2.2 Random Overwriting Viruses

Another rare variation of the overwriting method does not change the code of the program at the top of the host file. Instead, the virus seeks to a random location in the host program and overwrites the file with itself at that location. Evidently, the virus code might not even get control during execution of the host. In both cases , the host program is lost during the virus's attack and often crashes before the virus code can execute.

```
+----------------+
|PPPPPPPPPPPPPPPP|    clean program
+----------------+

+-----+---+------+
|PPPPP|VVV|PPPPPP|   infected program
+-----+---+------+
```

4.2.3 Appending Viruses

A common techniques introduced with DOS COM files is to place a jump (JMP) instruction at the front of the host to gain the execution flow. The jump instruction points to the virus code appended at the end of the original host file. The jump

```
+-----------+
|PPPPPPPPPPP|       clean program
+-----------+

+-+----------+---+
|V|PPPPPPPPPP|VVV|   infected program
+-+----------+^--+
|_____________|
    (jmp)
```

instruction is sometimes replaced with equally functional instructions, such as the following:

- CALL start_of_virus.
- PUSH offset start_of_virus
  ret.

Obviously the appender technique can be implemented for any other type of executable file, such as EXE, PE, ELF, Mach-O, and so on. Such files have in their header sections all the important values that can be used by the virus, such as the address of the main entry point.

4.2.4 Prepending Viruses

A common virus infection technique uses the principle of inserting virus code at the front of host programs. Such viruses are called *prepending viruses.*This is a simple kind of infection, and it is often very successful. Virus writer have implemented it on various operating system.
When a virus is written in assembler and it infects a COM file then it is simple for the virus to start the original host program. The virus copied the host code at the COM entry point (that is fixed at cs:0x100), and gives it control.
Prepender virus are often written in high-level languages such as C. Depending on the actual structure of the executable, the execution of the original program might not be as trivial task as it is for COM files. This is why a generic solution involves creation of a new temporary file on the disk to hold the content of the original host program. Then a sys-call, such as system(), is used to execute the original program in the temporary file.

```
+-----------+
|PPPPPPPPPPP|       clean program
+-----------+

+----+-----------+
|VVVV|PPPPPPPPPPP|   infected program
+----+-----------+
```

4.2.5 Classic Parasitic Viruses

A variation of the prepender technique is known as the classic parasitic infection. Such viruses overwrite the top of the host with their own code and save the top of the original host program to the very end of the host, usually virus-size long.
Some special parasitic infectors Instead of saving the top of the host to the end of it, use a temporary file to store this information outside of the file, sometimes with hidden attributes.

4.2.6 Cavity Viruses

Cavity viruses do not increase the size of the object they infect. Instead they overwrite a part of the file that can be used to store the virus code safely. Cavity infectors typically overwrite areas of files that contains zeroes or 0x20 (blank space).
A variation of cavity infection is called *fractioned cavity* technique. In this case the virus code is split between a loader routine and N number of section that contains blank spaces or zeroes. The loader (HEAD) routine of the virus locates the snippets of the virus code and read them into a continuous area of the memory.

4.2.7 Compressing Viruses

A special virus infection technique uses the approach of compressing the content of the host program. Sometimes this technique is used to hide the host's program size increase after the infection by packing the host program sufficiently with a binary packing algorithm.

4.2.8 Crypting Viruses

Some class of viruses can infect the host program with a crypted copy of its code. This technique is used to prevent antiviral software from detect them. A special class of these viruses can use a different class of crypting algorithm at every infection. So the virus can assume a different form each time it infect a program. This class of viruses is called *polymorphic viruses* and it is discussed more in detail later (**??**).
Crypting viruses must carry with them at each infection the decryptor routine, that has to decrypt the body of the virus before its execution.

4.2.9 Entry-Point Obscuring (EPO) Viruses

Entry point obscuring viruses do not change the entry point of the application to infect it, neither do they change the code at the entry point. Instead, they change the program code somewhere in such a way that the virus gets control randomly.

Function Call Hooking

A common technique of EPO viruses is to locate a function call reliably in the application's code section to a subroutine of the program. Because the pattern of a CALL instruction could be a part of another instruction's data, the virus would not be able to identify the instruction boundaries properly by looking for CALL instruction alone. To solve this problem, viruses often check to see whether the CALL instruction points to a pattern that appears to be the start of a typical subroutine call similar to the following:

```
          call Function
          ...

Function:
          push %ebp
          movl %esp,%ebp
```

API-Hooking Technique on Win32

On Win32 systems, EPO techniques became highly advanced. The PE file format can be attacked in different ways. One of the most common EPO techniques is based on the hooks of an instruction pattern in the program's code section. A typical Win32 application makes a lot of calls to APIs. Many Win32 EPO viruses take advantage of API CALL points and change these pointers to their own start code.
Once a CALL instruction is located in the host program's code section, the virus makes sure that it points to the import directory. In this way, the virus can reliably identify byte patterns that belong to a function call. After that, the CALL instruction is modified in such way that it will point to the start of the virus code located elsewhere, Typically appended to the end of the file. Such viruses typically search for one or both API call implementation:

Microsoft API Implementation	CALL DWORD PTR[]
Borland API Implementation	JMP DWORD PTR[]

The selection of functions to replace is often random, and the virus might not even get control each time a host program is executed. Viruses that want to execute everytime the host program runs can hook the ExitProcrss() API, that is called whenever the application exits back to the system. Normally disk activity increases whenever the application exits. This appends for several different reasons. For example, if an application has used a lot of virtual memory, the operating system will need to do a lot of paging, which increases disk activity. Thus it is likely that viruses like this remain unnoticed for a long time.

Import Table Replacing on Win32

Newer Win32 viruses infect Win32 executables in such a way that they do not need to modify the original code of the program to take control. To get control, the virus simply changes the import address table entries of the PE host in such a way that each API call of the application via the import address directory will run the virus code instead. The virus saves the original import address table entries so that after its own code execution it can call properly the API originally wanted by the host application.

Chapter 5

Classification of In-Memory strategies

5.1 Direct-Action Viruses

Some of the simplier viruses do not remain active in memory. They simply load with the host program and perform infection before or after the host program has executed. Direct-action viruses typically use a FindFirst-FindNext sequence to look for a set of victim application to attack. Typically such viruses only infect a couple of files upon execution otherwise the user can notice an overload of disk activity.

5.2 Memory-Resident Viruses

A much efficient class of computer viruses remains in memory after the initialization of virus code. Such viruses typically follow these steps:

- The virus gets control of the system
- It allocates a block of memory for its own code
- It relocates its code to the allocated block of memory
- It activates itself in the allocated memory block
- It hooks the execution of the code flow to itself
- It infects new files and/or system areas

This is the most typical pattern, but several other methods exist that do not require all of the preceding steps.
On single-tasking operating systems such as DOS, only a single-user application can run at any one time; any other program code needs to make itself TSR (Terminate and Stay Resident).
A common technique to gain the execution of the code flow on DOS, is to hook one

INT ID	Function/Category	intercepted/Used by Virus Code
int 00	Divide Error/CPU generated	Antidebugging, Anti-Emulation
int 01	Single Step/CPU generated	Ante-Debbugging, EPO
int 03	Breakpoint/CPU generated	Anti-Debugging, Tracing
int 08	System Timer/CPU generated	Activation routine, Anti-debugging
int 09	Keyboard/BIOS	Anti-debugging, password stealing
int 1C	System Timer Tick/BIOS	Activation routine
int 20	Terminate Program/DOS Kernel	Infect on Exit
int 21	DOS Service/DOS Kernel	Infection, Stealth, Activation routine
int 25	Absolute Disk Read/DOS Kernel	Disk Infection, Stealth
int 26	Absolute Disk Write/DOS Kernel	Disk Infection, Stealth
int 27	TSR/DOS Kernel	Remain in memory
int 28	IDLE interrupt/DOS Kernel	to perform an action while idle

Table 5.1: Typical Interrupts Used by Computer Viruses

or more interrupts. A virus can hook the timer interrupt (int 0x1C) and take control periodically.
The boot viruses used to hook the int 0x13 disk interrupt handler and start to monitor its functions, wait for diskette access for read and write, and during such operation write their code (or part of it) into the boot sector of the diskettes.
Table 5.1 shows common interrupts and their typical use by computer viruses.

5.3 Stealth Viruses

Stealth Viruses always intercept a single function or a set of functions in such a way that the caller of the function will receive manipulated data from the hook routine of the virus. One of the first-known viruses on the PC, Brain (a boot virus), was already stealth. Brain showed the original boot sector whenever an infected sector was accessed and the virus was active in memory. The virus hooked the int 13h and returned manipulated data to the interrupt caller.
The stealth technique also quickly appeared in DOS file infector viruses. This method was a sure way for a virus to go unnoticed for a relatively long period of time. In fact, in the DOS days, users would remember sizes of system files in an attempt to apply their own integrity checking. By knowing the original size of a file such as COMMAND.COM, the commander interpreter was halfway to success in finding an on-going infection.
We call a virus *semistealth* if it hides the change of the file size but the changed content of the infected objects remains visible via regular file access.

5.4 Viruses in Processes (in User Mode)

On modern multitasking operating systems, viruses need to use slightly different strategies. The virus code does not have to become "resident" in the traditional sense. It is usually enough if the virus runs itself as a part of the process. Memory space is divided into security rings. Ring zero is assigned the the system kernel and have plain control of the machine. User's program runs on the last level (four) and have the lowest security level. Thus application normally do not interfere with the system kernel, as DOS programs do.
An attacker has several options:

- The virus loads with the infected process, gets control, creates a thread (or a set of threads) in the running process itself in the user mode, and infects files using regular direct-action technique.
- a virus can run in its own process in user mode. It can **fork**, and remain active in memory in an infinite loop.

On UNIX systems the file system is designed for security and a virus that runs in user-mode can access another file only if it has the right privileges.

5.5 Viruses in Kernel Mode

When a virus gains ring zero with privilege escalation, it has the total control of the machine. Now it can alter all the system files and the infection of the system is done on its roots. If a stealth technique is used from ring zero then we have the so called ROOTKIT. The first generation of rootkits modified some strategical system files so that viral files of viral effects resulted invisible to any user. In Unix systems the first generation of rootkits simply substitute critical system programs that listed processes, network connections and filesystem data such as ps, netstat,and ls. But if a user (or superuser) executed a clean version of those programs (or other similar programs) from a clean boot, all the hidding system is bypassed and the viral attack can be discovered.
The second generation of rootkits attacked directly the kernel hooking directly system call to modify users request that can reveal the system infection such as linting processes, listing network connections, and filesystem data. Thus any program is launched by the corrupted kernel, it returns altered data.

Chapter 6

Advanced Virus Techniques

6.1 Armored Viruses

A computer virus's primary goal is to spread as far as possible without being noticed. The authors of armored viruses want to be sure that the virus code is difficult for scanners to detect, even if scanners use techniques such as heuristic that can pinpoint previously unknown computer viruses. Furthermore, if a virus sample is obtained by any means, its author wants to make the analysis of the virus code as difficult as possible to further delay rapid response to the virus attack.
The following sections describe basic methods of armored viruses.

6.1.1 Antidisassembly

Computer viruses written in Assembly language are challenging to understand because they often use tricks that normal programs never or very rarely use.
One of the most obvious ways to avoid disassembling is to use encrypted data in the virus code. All content data in the virus can be encrypted. When the virus is loaded into the disassembler, the virus code will reference encrypted snippets, which you need to decrypt one by one to understand what the virus does with them, making virus analysis en even more tedious process.
Another possibility for the attacker to challenge disassembling is to use some sort of self-modifying code. When the code is examined in the disassembler, it might not be easily read (Figure 6.1).

```
movl $0x100,%ecx              movl $0x3f,%ecx
movw $0x40,%ah                incl %ecx
int $0x21                     xchgw %cl,%ch
                              xchl %ecx,%eax
                              movl $0x100, %ecx
                              int $0x21
```

Figure 6.1: Example of slightly obfuscated code.

Another technique to trick old disassemblers was to put a single data byte into the code so that the disassembler would get confused interpretating the opcodes (Figure 6.2).

```
pusha                                pusha
call near ptr loc_4013E6+1           call loc_4013E7
loc_4013E6:                          db 0B8h
mov eax, 0B1C9335Bh                  loc_4013E7:
add bl,ah                            pop ebx
dec ecx                              xor ecx,ecx
mov byte ptr [ebx+5], 0              mov cl,0
lea edi, [ebx+400h]                  jecxz short near ptr unk_401437
mov al,[ebp+10h]                     mov byte ptr [ebx+5], 0
dec al                               les edi, [ebx+400h]

(opcode mixing)                      (correctly disassembled code)
```

Figure 6.2: Opcode mixing code confusion

In the example above the CALL/POP pair ensures the correct code execution, but old disassemblers get confused because inserted the 0xB8 (MOV) opcode into the code flow.
A CALL to a POP instruction is a common sequence in computer viruses to adjust for their location in the file.

6.1.2 Antidebugging

The debugging activity is about to execute the code step by step or to execute a portion of code via a breakpoint for evaluating and inspect registers or variables. So a common technique to avoid debugging is to hook strategical interrupts such as int1 and int3. Typically those interrupts point to a simple IRET routine. A virus can hook these interrupts to do other things such as decrypt their body, or execute next instruction. This will completely by-pass the debugger.
On Windows 9x the interrupt descriptor table (IDT) can be manipulated from user

mode and some virus, like the *W95/CIH virus* did, used this technique to jump to kernel mode ans at the same time avoided the debugger.
Some set of virus, like *Whale virus*, can take advantage from intrinsic characteristic of hardware such as the **prefetch-queue** of the CPU. With this method a virus can change the opcode of an instruction that was already prefetched by the processor, confusing the debugger.
When you a are tracing a code, you are using the keyboard. A very brutal antidebugging technique is to disable the keyboard to prevent you from continuing to trace, perhaps by reconfiguring the content of the I/O PORT 0x21 or PORT 0x61. Such ports might be different according to how modern operating system map them.

6.1.3 Antiheuristics

In 1998, Windows virus development was in a relatively early stage. This is why a wide variety of different infection methods were introduced, making it possible to consider heuristic analysis against 32-bit Windows viruses. *Heuristic analysis* can detect unknown viruses and closely related variants of existing viruses using static and dynamic methods. *Static heuristic* rely on file format and common code fragment analysis. *Dynamic heuristic* use code emulation to mimic the processor and the operating system environment and detect suspicious operations as the code is "running" in the virtual machine of the scanner.
Careful analysis of different infection types led to the development of first generation Win32 heuristic detectors, which used static heuristics. Static heuristics are capable of pinpointing suspicious portable executable (PE) file structures and therefore can catch first-generation 32-bit Windows viruses with a very high detection rate. The idea was based on DOS virus detection that uses similar methods.
By the end of 1999, many new virus replication methods had already been developed. Moreover, a major part of 32-bit Windows viruses used some sort of encryption, polymorphic, or metamorphic technique. Encrypted viruses are more difficult to detect and vary dangerous when we look ahead to the future of scanning. This is because scanners can become very slow in attempting decryption for too many clean files on your system.
First-generation heuristics were extremely successful against PE file viruses. Even virus writers were surprised by their success, but we did not have to wait long before they came up with attacks against heuristic detectors.
We describe below attacks against first-generation Win32 heuristic.

New PE File-Infection Techniques

Many PE file viruses add a new section or append to the last section of PE files. This heuristic has the benefit that can be performed easily by end user. The heuristic checks to see if the entry point of the PE file points to the last section of the application.
A large number of commercial PE file on-the-fly packers were introduced for Win32

platforms, such as UPX, Neolite, Petite, Shrink32, ASPack, and so on. Such packers are commonly used by virus writers and Trojan code creators to hide their creations. They can use advanced antiheuristics techniques such as encrypting suspicious string or insert code or section not at the last position.

More Than One Virus Section

Several Win32 viruses append not only to one section but to many sections at a time. For instance the W32/Resure virus appends four sections (.text, .rdata, .data, and .reloc) to each PE host. Because the entry point of the application will be changed to point into the new .text section of the virus and because it will not be the last section, the heuristic fools.

Entry-Point Obscuring

This is one of the most powerful antiheuristic of ever. This technique was already discussed in earlier chapters(4.2.9).

No CALL-to-POP Trick

Most 32-bit Windows appending viruses use the CALL-to-POP trick to locate the start address for their data relocations. Because first-generation heuristics could easily look for that, virus writers tried to implement new ways to get the base address of the code.
A common way is to use the opcode mixing code confusion technique (Figure 6.2).

6.1.4 Antiemulation

Dynamic heuristic use code emulation to mimic the processor and the operating system. Some virus writer introduced antiemulation techniques against the strongest component of the antivirus product: the emulator.
We describe below some of the antiemulation techniques that have been used by virus writers over the years.

Using the Coprocessor (FPU) Instructions

Some virus writers realized the power of emulators and looked for weaknesses and quickly realized that coprocessor emulation was not implemented. Infact most emulators skipped coprocessor instructions until recently, whereas most processors that are currently used support coprocessor instruction by default.

Using MMX Instruction

Other virus writers went so far as to implement a virus that used the MMX (multimedia extension) instructions of Pentium processors.
Even these instructions where not implemented in antiviral emulator.

Using Undocumented CPU Instructions

Although there are not too many undocumented Intel processor instructions, there are a few. For example, W95/Vulcano virus uses the undocumented SALC instruction in its polymorphic decryptor as garbage to stop the processor emulators of certain antivirus that cannot handle it. Intel claims that SALC can be emulated as a NOP (a no-operation instruction). Hardly a NOP, this instruction sets AL=FFh if the carry flag is set (CF=1), or resets AL to zero if the carry flag is clear(CF=0). Some emulators' implementation of these instruction might differ subtly from the processor's so that a virus could detect when it is executing under emulation.

Using Brute-Force Decryption of Virus Code

Some viruses, use a brute-force algorithm also known as RDA (Random Decryption Algorithm) to decrypt themselves. This method was used by DOS viruses, such as Spanska virus variants and some older Russian viruses, such as the RDA family. All of these old tricks of virus writers have been recycled in Win32 viruses. Brute-force decryption does not use fixed keys but tries to determine the actual method and the proper keys by trial and error. This logic is relatively fast in the case of real-time execution, but it generates very long loops causing zillions of emulation iterations, ensuring that the actual virus body will not be reached easily.

Using Multithreaded Virus Functionality

Many viruses tried to use threads to give emulators a hard time. Emulators were first used to emulate DOS applications. DOS only supported single-threaded execution, a much simpler model for emulators than the multithreaded model.
Emulation of multithreaded Windows applications is challenging because the synchronization of various threads is crucial but rather difficult.

6.1.5 Aggressive Retroviruses

A *retrovirus* is a computer virus that specifically tries to bypass or hinder the operation of an antivirus, personal firewall, or other security programs.
There are many possible way for an attacker to achieve this because most Windows users work with their administrative privileges. This give computer viruses the potential to kill the processes and files that belong to antivirus software or to disable antivirus programs.

Retroviruses have the potential to make way for other computer viruses that are otherwise known and easy for the antivirus software to handle. Therefore, virus writers routinely reverse engineer antivirus products to learn tricks that can be used in retro attacks.

6.2 Polymorphic Viruses

6.2.1 Encrypted Viruses

From the very early days, virus writers tried to implement virus code evolution. One of the easiest ways to hide the functionality of the virus code was *encryption.* The first known virus that implemented encryption was Cascade on DOS. The virus starts with a constant decryptor, which is followed by encrypted virus body. Consider the example extracted from Cascade.1701 shown in Figure 6.3.

```
lea si, Start ; position to decrypt (dynamically set)
mov sp, 0682  ; lenght of encrypted body

Decrypt:
xor [si],si   ; decryption key/counter 1
xor [si],sp   ; decryption key counter 2
inc si        ; increment one counter
dec sp        ; decrement the other
jnz Decrypt   ; loop until all bytes are decrypted

Start:        ; Encrypted/Decrypted Virus Body
```

Figure 6.3: The Decryptor of the Cascade Virus.

Note that this decryptor has antidebugging features because the SP (stack pointer) register is used as one of the decryption keys. The direction of the decryption loop is always forward; the SI register is incremented by one.
Cascade appends itself to the files, so SI will result in the same value if the two hosts have the same size. However, the SI (decryption key 1) is changed if the host programs have different size. The core of the encryption algorithm consists of only two XOR instruction. XOR is very practical for viruses because XORing with the same value twice results in the initial value.
Cryptographically speaking, such encryption is weak, though early antivirus programs had little choice but to pick a detection string from the decryptor itself. This led to a number of problems, however. Several different viruses might have the same decryptor, but might have completely different functionalities. By detecting the virus based on its decryptor, the product in unable to identify the variant or the virus itself.

More importantly, non-viruses, such as antidebugging wrappers, might have a similar decryptor in front of their code. As a result, the virus that uses the same code to decrypt itself will confuse them.
The attacker can implement more complicated, further confusing the antivirus program's detection and repair routines:

- The direction of the loop can change.
- Multiple layers of encryption are used. The first decryptor decrypts the second one, the second decrypts the third, and so on.
- several encryption loops take place one after another, with randomly selected directions.
- There is only one decryption loop, but is uses more than two keys to decrypt each encrypted piece of information on the top of the others.
- Start of the decryptor is obfuscated.
- Nonlinear decryption is used.

6.2.2 Oligomorphic Viruses

Unlike encrypted viruses, *oligomorphic viruses* do change their decryptors in new generations. The simplest technique to change the decryptor is to use a set of decryptors instead of a single one. The first known virus to use this technique was Whale. Whale carried a few dozen different decryptors, and the virus picked one randomly.
W95/Memorial had the ability to build 96 different decryptor patterns. Thus the detection of the virus based on the decryptor code was an impractical solution, though a possible one. Most products tried to deal with the virus by dynamic decryption of the encrypted code. The detection is still based on the constant code of the decrypted virus body.
In Figures 6.4 and 6.5 there are shown two variant of the Memorial virus decryptor. Notice the appearance of a "loop" instruction in this instance, as well as the swapped instructions in the front of the decryptor. A virus is said to be oligomorphic if it is capable of mutating its decryptor only slightly.

6.2.3 Polymorphic Viruses

Polymorphic viruses can mutate their decryptors to a high number of different instances that can take millions of different forms.
The first known polymorphic virus is the *1260 virus*. The virus uses two sliding keys to decrypt its body, but more importantly, it inserts junk instructions into its decryptor. These instructions are garbage in the code. They have no function other than altering the appearance of the decryptor.

```
mov ebp,00405000h          ; select base
mov ecx,0550h              ; this many bytes
lea esi,[ebp+0000002E]     ; offset of "Start"
add ecx,[abp+00000029]     ; plus this many bytes
mov al,[ebp+0000002D]      ; pick the first key

Decrypt:
nop                        ; junk
nop                        ; junk
xor [esi],al               ; decrypt a byte
inc esi                    ; next byte
nop                        ; junk
inc al                     ; slide the key
dec ecx                    ; are there any more bytes to decrypt?
jnz Decrypt                ; until all bytes are decrypted
jmp Start                  ; decryption done, execute body

;Data area

Start
; encrypted/decrypted virus body
```

Figure 6.4: An Example Decryptor of the W95/Memorial Virus.

```
mov ecx,0550h               ; this many bytes
mov ebp,013BC000h           ; select base
lea esi,[ebp+0000002E]      ; offset of "Start"
add ecx,[abp+00000029]      ; plus this many bytes
mov al,[ebp+0000002D]       ; pick the first key

Decrypt:
nop                         ; junk
nop                         ; junk
xor [esi],al                ; decrypt a byte
inc esi                     ; next byte
nop                         ; junk
inc al                      ; slide the key
loop Decrypt                ; until all bytes are decrypted
jmp Start                   ; decryption done, execute body

;Data area

Start
; encrypted/decrypted virus body
```

Figure 6.5: A Slightly Different Decryptor of the W95/Memorial Virus.

Virus scanners were challenged by 1260 because simple search strings could no longer be extracted from the code. Although 1260's decryptor is very simple, it can become shorter or longer according to the number of inserted junk instructions and random padding after the decryptor for up to 39 bytes of junk instructions. In addition, each group of instructions (prolog, decryption, and increments) within the decryptor can be permutated in any order. Thus the "skeleton" of the decryptor can change as well. In Figure 6.6 there is an extract of one of the variations of the 1260 virus. In each group of instructions, up to five junk instructions are inserted (INC SI, CLC, NOP, and other do-nothing instructions) with no repetitions allowed in the junk. There are two NOP junk instructions that always appear.

```
; Group 1 - Prolog Instructions
inc si                  ;optional, variable junk
mov ax,0E9B             ; set key 1
clc                     ; optional variable junk
mov di,012A             ; offset of Start
nop                     ; optional, variable junk
mov cx,0571             ; this many bytes - key 2

; Group 2 - Decryption Instructions
Decrypt:
xor [di],cx            ; decrypt first word with key 2
sub bx,dx              ; optional, variable junk
xor bx,cx              ; optional, variable junk
sub bx,ax              ; optional, variable junk
sub bx,cx              ; optional, variable junk
nop                    ; non-optional junk
xor dx,cx              ; optional, variable junk
xor [di],ax            ; decrypt first word with key 1

; Group 3 - Decryption Instructions
inc di                 ; next byte
nop                    ; non-optional junk
clc                    ; optional, variable junk
inc ax                 ; slide key 1
loop Decrypt           ; until all bytes are decrypted - slide key 2
;random padding

Start:
; encrypted/decrypted virus body
```

Figure 6.6: An Example Decryptor of 1260 virus.

Part II

Emulation Environments

Chapter 7

Viruses and Artificial Life

In this chapter, we examine the question of whether a virus represents or not a form of artificial life. The first, and obvious, question is "What is life?". Without an answer to this question, we will be unable to say if a computer virus is "alive." One very reasonable list of properties associated with life was presented in the list below:

- Life is a pattern in space-time rather than a specific material object.
- Self-reproduction, in itself or in a related organism.
- Information storage of a self-representation.
- A metabolism that converts matter/energy.
- Functional interactions with the environment.
- Interdependence of parts.
- Stability under perturbations of the environment.
- The ability to evolve.
- Growth or expansion.

Let us examine each of these characteristics in relation to computer viruses.

7.1 Viruses as patterns in space-time

There is a near match to this characteristic. Viruses are represented by patterns of computer instructions that exist over time on many computer systems. Viruses are not associated with the physical hardware, but with the instructions executed (sometimes) by that hardware. Computer viruses, like all functional computer code, are simply manifestations of algorithms. The algorithms themselves also represent an underlying pattern.

It is questionable if these patterns exist in space, however, unless one extends the definition of space to "cyberspace" as represented by a computer system. The patterns of the viruses are a temporary set of electrical and magnetic field changes in the memory or storage of computer systems. The existence of the virus is only within these patterns of energy. Arguably, the code for each virus could be printed in ink on paper, resulting in a more substantiative existence. That, however, is merely a representation of the true virus, and should not be viewed as existence any more than a picture of a person is itself the person.

7.2 Self-reproduction of viruses

One of the primary characteristics of computer viruses is their ability to reproduce themselves (or an altered version of themselves). Thus, this characteristic seems to be met. One of the key characteristics is their ability to reproduce.
However, it is perhaps more interesting to examine this aspect in light of the agent of reproduction. The virus code is not itself the agent; the computer is. It is questionable if this can be considered sufficient for purposes of classification as artificial life. To do so would imply that (for instance) the blueprints for a Xerox machine are capable of self-reproduction: when outside agents follow the instructions therein, it is possible to produce a new machine that can then be used to make a copy of them. It is not the blueprint (algorithm; virus) that is the agent of change, but the entity that interprets it.

7.3 Information storage of a self-representation

This is the most obvious match for computer viruses. The code that defines the virus is a template that is used by the virus to replicate itself. This is similar to the DNA molecules of what we recognize as organic life.

7.4 Virus metabolism

This property involves the organism taking in energy or matter from the environment and using it for its own activity. Computer viruses use the energy of computation expended by the system to execute. They do not convert matter, but make use of the electrical energy present in the computer to traverse their patterns of instructions and infect other programs. In this sense, they have a metabolism.
Again, however, we are forced to change this view if we examine the case more closely. The expenditure of energy is not by the virus, but by the underlying computer system. If the virus were not active, and an interactive game were being run instead, the same amount of energy would be used. In most systems, even if no program is being run,

the energy use remains constant. Thus, we must conclude that viruses do not actually have a metabolism.

7.5 Functional interactions with the viruses environment

Viruses perform examinations of their host environments as part of their activities. They alter interrupts, examine memory and disk architectures, and alter addresses to hide themselves and spread to other hosts. They very obviously alter their environment to support their existence. Many viruses accidentally alter their environment because of bugs or unforeseen interactions. The major portion of damage from all computer viruses is a result of these interactions.

7.6 Interdependence of virus parts

Living organisms cannot be arbitrarily divided without destroying them. The same is true of computer viruses. Should a computer virus have a portion of its anatomy excised, the virus would probably cease to function normally, if at all. Few viruses are written with superfluous code, and even so, the working code cannot be divided without disabling the virus.
However, it is interesting to note that the virus can be reassembled later and regain its functional status. If a living organism (as we know them) were to be divided into its component parts for a period of time, then reassembled, it would not become "alive" again. In this sense, computer viruses are more like simple machines or chemical reactions rather than instances of living things.

7.7 Virus stability under perturbations

Computer viruses run on a variety of machines under different operating systems. Many of them are able to compromise (and defeat) anti-virus and copy protection mechanisms. They may adjust on-the-fly to conditions of insufficient storage, disk errors, and other exceptional events. Some are capable of running on most variants of popular personal computers under almost any software configuration; a stability and robustness seen in few commercial applications.

7.8 Virus evolution

Here, too, viruses display a difference from systems we traditionally view as alive. No computer viruses evolve as we commonly use the term, although it is conceivable that a very complex virus could be programmed to evolve and change. However, such a

virus would be so large and complex as to be many orders of magnitude larger than most host programs, and probably bigger than the host operating systems. Thus, there is some doubt that such a virus could run on enough hosts to allow it to evolve. (Note that "evolve" implies a change in function or attributes; polymorphic viruses represent cases of random changes in structure but not functionality.)
Higher-level mutations of viruses do exist, however. There are variants of many known viruses, with over a dozen known for some IBM PC viruses. The variations involved can be very small, on the order of two or three instructions difference, to major changes involving differences in messages, activation, and replication. The source of these variations appears to be programmers (the original virus authors or otherwise) who alter the viruses to avoid anti-viral mechanisms, or to cause different kinds of damage. Polymorphic viruses alter their copies to avoid detection, but the pattern of alteration is ultimately a human product. These changes do not constitute evolution, however.
Interestingly, there is also one case where two different strains of a Macintosh virus are known to interact to form infections unlike the parents, although these interactions usually produce sterile offspring that are unable to reproduce further. This likewise does not appear to be evolution as we know it.

7.9 Growth

Viruses certainly do exhibit a form of growth, in the sense that there are more of them in a given environment over time. Some transient viruses will infect every file on a system after only a few activations. The spread of viruses through commercial software and public bulletin boards is another indication of their wide-spread replication. Although accurate numbers are difficult to derive, reports over the last few years indicate an approximate yearly doubling in the number of systems infected by computer viruses. Clearly, computer viruses are exhibiting significant growth.

7.10 Other behavior

As already noted, computers viruses exhibit species with well-defined ecological niches based on host machine type, and variations within these species. These species are adapted to specific environments and will not survive if moved to a different environment.
Some viruses also exhibit predatory behavior. For instance, the DenZuk virus will seek out and overwrite instances of the Brain virus if both are present on the same system. Other viruses exhibit territorial behavior, marking their infected domain so that others of the same type will not enter and compete with the original infection. Some viruses also exhibit self-protective behavior, including camouflage techniques.
It is important to note, however, that none of these characteristics came from the

viruses themselves. Rather, each change and addition to virus behavior has been wrought by an outside agency: the programmer. These changes have been in reaction to a perceived need to "enhance" the virus; usually to make it more difficult to find. It might well be argued that more traditional living organisms may also undergo change from without. As an example, background radiation may cause occasional random mutations. However, programmers are the only source of change to computer viruses, and this distinction is worth noting; other living systems undergo changes to themselves and their progeny without obvious outside agencies.

7.11 Concluding Comments

Our study of computer viruses at first suggests they are close to what we might define as "artificial life." However, upon closer examination, a number of significant deficiencies can be found. These lead us to conclude that computer viruses are not "alive," nor is it possible to refine them so as to make them "alive" without drastically altering our definition of "life."

Undoubtedly, we could adjust our definitions and characteristics to encompass computer viruses or to better exclude them. This illustrates one of the fundamental difficulties with the entire field of artificial life: how to define essential characteristics in such a way as to unambiguously define living systems. Computer viruses provide one interesting example against which such definitions may be tested.

From this, we can observe that computer viruses (and their kin) provide an interesting means of modeling life. For at least this reason, research into computer viruses may be of some scientific interest. By modeling behavior using computer viruses, we may be able to gain some insight into systems with more complex interactions. Research into competition among computer viruses and other software, including anti-viral techniques, is of practical interest as well as scientific interest. Modified versions of viruses such as Thimbleby's Liveware may also prove to be of ultimate value. Research into issues on virus defense methods, epidemiology, and on mutations and combinations also could provide valuable insight into computing.

The problem with research on computer viruses is their threat. True viruses are inherently unethical and dangerous. They operate without consent or knowledge, experience has shown that they cannot be recalled or controlled, and they may cause extensive losses over many years. Even viruses written to be benign cause significant damage because of unexpected interactions and bugs.

The origin of most computer viruses is one of unethical practice. Viruses created for malicious purposes are obviously bad; viruses constructed as experiments and released into the public domain would likewise be unethical, and poor science besides: experiments without controls, strong hypotheses, and the consent of the subjects. Facetiously, I suggest that if computer viruses evolve into something with artificial consciousness, this might provide a doctrine of "original sin" for their theology.

Computer viruses have caused millions of dollars of damage and untold aggravation.

Some of them have been written as harmless experiments that "got away," and others as malicious mischief. A great many of them have firmly rooted themselves in the pool of available computers and storage media, and they are likely to be frustrating users and harming systems for years to come. Similar but considerably more tragic results could occur from careless experimentation with organic forms of artificial life. We must never lose sight of the fact that "real life" is of much more importance than "artificial life," and we should not allow our experiments to threaten our experimenters.

Chapter 8

The WiCE Language

The WiCE language is derived from the 1994 Red Code standard (ICWS'94). Red Code is a language used for many emulators such as CoreWars. The language is very simple. It is a kind of Assembly language without no registers. The data are read, manipulated and stored into the memory. The early '88 standard of Red Code was made of a few basic instructions with only direct, immediate and indirect address mode and with no instruction modifier. One of the most important things that the '94 standard of Red Code brought was modifiers and new addressing modes. In the old '88 standard the addressing modes alone decide which parts of the instructions are affected by an operation. For example, MOV 1, 2 always moves a whole instruction, while MOV #1, 2 moves a single number. (and always to the B-field!).
WiCE differs only a little on the ICWS'94 syntax. The only differences are listed in Table 8.1.

Instruction type	ICWS'94	WiCE Language
assert directive	;*assert <expression>*	*assert <expression>*
labels	*label*	*label*:
for/rof construct	implemented	not yet implemented

Table 8.1: Differences between ICWS'94 and WiCE language.

The *assert* directive has the semicolon in ICWS'94 like any other comment, but in WiCE language *assert* is not preceded bay any semicolon.
Labels are ended bay a colon in WiCE language. Indeed in the ICWS'94 they are ended by a blank or space character.
And last the *for/rof* construct in not yet been implemented on WiCE language. Maybe it will in next versions.

8.1 The Grammar

The grammar used by the WiCE language is listed below:

```
list-->          line | line list
line-->          comment | instruction | directive
comment-->       ; v* EOL | EOL
directive-->     assert_d | org_d | equ_d
assert_d-->      ASSERT bexpr
org_d-->         ORG label
equ_d-->         label EQU expr
instruction-->   label_list operation mode expr comment |
                 label_list operation mode expr , mode expr comment
label_list-->    label | label label_list | label newline label_list | e
label-->         alpha alphanumeral* :
operation-->     opcode | opcode.modifier
opcode:          DAT | MOV | ADD | SUB | MUL | DIV | MOD |
                 JMP | JMZ | JMN | DJN | CMP | SLT | SPL |
                 ORG | EQU | END | CPIN | CTIN | CPOUT | CTOUT
modifier-->      A | B | AB | BA | F | X | I
mode-->          # | $ | @ | < | > | { | } | e
bexpr-->         expr < expr | expr == expr | expr <= expr |
                 expr != expr | expr
expr-->          term |
                 term + expr | term - expr |
                 term * expr | term / expr |
                 term % expr
term-->          label | number | (expression)
number-->        whole_number | signed_integer
signed_integer-->+whole_number | -whole_number
whole_number-->  numeral+
alpha-->         A-Z | a-z | _
numeral-->       0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
alphanumeral-->  alpha | numeral
v-->             ^EOL
EOL-->            newline | EOF
newline-->        LF | CR | LF CR | CR LF
e-->
```

8.2 Run-time Variables

In addition to the variables defined in its own code, every warrior can access a set of global variables created by the WiCE environment. These global variables are injected in compile-time into the table of variables of each warrior.
The global variables refers to the actual environment in which the warrior runs.
Run-time variables consist of the following:

CORESIZE :	the size of the memory array. The default is 8000.
WARRIORS :	the initial number of warriors to battle simultaneously in the arena.
MAXPROCESSES :	each warrior can spawn multiple additional threads. This variable sets the maximum number of thread allowed per warrior. the default is 1000.
MAXCYCLES :	this is the maximum CPU cycles. After that the battle is declared as a tie. the default is 10000.
MAXLENGTH :	this is the maximum lenght in terms of instruction code allowed by each warrior (the size of the warrior). The default is 200.
MINDISTANCE :	this is the minimum distance between warriors when they are allocated in the arena by the emulator. The default is 200.
VERSION :	this is the version number of the WiCE emulator.

8.3 General Definitions

- An instruction consists of an opcode, a modifier, an A-operand, and a B-operand.
- An A-operand consists of an A-mode and an A-number.
- An A-mode is the addressing mode of an A-operand.
- An A-number is an integer between 0 and M-1, inclusive.
- A B-operand consists of a B-mode and a B-number.
- A B-mode is the addressing mode of a B-operand.
- A B-number is an integer between 0 and M-1, inclusive.

8.4 Specific Definitions

- The program counter (PC) is the pointer to the location in core of the instruction fetched from core to execute. Some times the program counter is called Instruction Pointer (IP).
- The current instruction is the instruction in the instruction register, as copied (prior to execution) from the PC location of core.
- The A-pointer points to the instruction the A-operand of the current instruction references in core.
- The A-instruction is a copy of the instruction the A-pointer points to in core (as it was during operand evaluation).
- The A-value is the A-number and/or the B-number of the A-instruction or the A-instruction itself, whichever are/is selected by the opcode modifier.
- The B-pointer points to the instruction the B-operand of the current instruction references in core.
- The B-instruction is a copy of the instruction the B-pointer points to in core (as it was during operand evaluation).
- The B-value is the A-number and/or the B-number of the B-instruction or the B-instruction itself, whichever are/is selected by the opcode modifier.
- The B-target is the A-number and/or the B-number of the instruction pointed to by the B-pointer or the instruction itself, whichever are/is selected by the opcode modifier.

8.5 Instruction Set

The number of instructions in Redcode has grown with each new standard, from the original number of about 5 to the current 18 or 19. And this doesn't even include the new modifiers and addressing modes that allow literally hundreds of combinations. Luckily, we don't need to learn all the combinations. It is enough to remember the instructions, and how the modifiers change them.
Here is a list of all the instructions used in WiCE language:

- DAT – data (kills the process)
- MOV – move (copies data from one address to another)
- ADD – add (adds one number to another)

- SUB – subtract (subtracts one number from another)
- MUL – multiply (multiplies one number with another)
- DIV – divide (divides one number with another)
- MOD – modulus (divides one number with another and gives the remainder)
- JMP – jump (continues execution from another address)
- JMZ – jump if zero (tests a number and jumps to an address if it's 0)
- JMN – jump if not zero (tests a number and jumps if it isn't 0)
- DJN – decrement and jump if not zero (decrements a number by one, and jumps unless the result is 0)
- SPL – split (starts a second process at another address)
- CMP – compare (same as SEQ)
- SEQ – skip if equal (compares two instructions, and skips the next instruction if they are equal)
- SNE – skip if not equal (compares two instructions, and skips the next instruction if they aren't equal)
- SLT – skip if lower than (compares two values, and skips the next instruction if the first is lower than the second)
- LDP – load from p-space (loads a number from private storage space)
- STP – save to p-space (saves a number to private storage space)
- NOP – no operation (does nothing)
- CTIN – Communication between Threads IN channel.
- CTOUT – Communication between Threads OUT channel.
- CPIN – Communication between Processes IN channel.
- CPOUT – Communication between Processes OUT channel.

All WiCE instructions are executed following the same procedure:

1. The currently executing warrior's current task pointer is extracted from the warrior's task queue and assigned to the program counter.

2. The corresponding instruction is fetched from core and stored in the instruction register as the current instruction.
3. The A-operand of the current instruction is evaluated.
4. The results of A-operand evaluation, the A-pointer and the A-instruction, are stored in the appropriate registers.
5. The B-operand of the current instruction is evaluated.
6. The results of B-operand evaluation, the B-pointer and the B-instruction, are stored in the appropriate registers.
7. Operations appropriate to the opcode.modifier pair in the instruction register are executed. With the exception of DAT instructions, all operations queue an updated task pointer. (How the task pointer is updated and when it is queued depend on instruction execution).

8.5.1 pseudo-instructions

"ORG" ("ORiGin") is a way for the source assembly file to indicate the logical origin of the warrior. The A-operand contains an offset to the logical first instruction. Thus "ORG 0" means execution should start with the first instruction (the default) whereas "ORG 6" means execution should start with the seventh instruction."ORG *label*" is allowed also and the label's address is solved on the second pass of the parser. Although multiple ORG instructions are of no additional benefit to the programmer, they are allowed. When there is more than one ORG instruction in a file, the last ORG instruction encountered will be the one that takes effect.

"EQU" ("EQUate") is a simple text substitution utility. Instructions of the form "label EQU expression" will replace all occurrences of "label" with the equivalent solution of the expression.

The "ASSERT" is followed by a logical expression. If it's false, the program will not be compiled. In C, a value of 0 means false and anything else means true. The logical and comparison operators return 1 for true, a fact which can be useful later. It can be used to make sure the program really works with the current settings. Typically, "ASSERT" is used to check that the size of the core is the one the constants have been designed for, like "assert CORESIZE == 8000".

"END" indicates the logical end of the assembly file. If an END instruction is found by the parser, the the parsing process exits and all eventually instructions that follows the END won't be compiled.

8.5.2 DAT

It has two purpose. The first is to store program's data. The second is to kill the process that executes this instruction. The execution of a DAT effectively removes the currently executing process from the process queue.

8.5.3 MOV

Move replaces the B-target with the A-value and queues the next instruction (PC + 1). MOV is one of the few instructions that support .I, and that's its default behavior if no modifier is given (and if neither of the fields uses immediate addressing).

8.5.4 ADD

ADD replaces the B-target with the sum of the A-value and the B-value (A-value + B-value) and queues the next instruction (PC + 1). ADD.I functions as ADD.F would.
Notice that all math in WiCE is done **modulo CORESIZE**.

8.5.5 SUB

SUB replaces the B-target with the difference of the B-value and the A-value (B-value - A-value) and queues the next instruction (PC + 1). SUB.I functions as SUB.F would.

8.5.6 MUL

MUL replaces the B-target with the product of the A-value and the B-value (A-value * B-value) and queues the next instruction (PC + 1). MUL.I functions as MUL.F would.

8.5.7 DIV

DIV replaces the B-target with the integral result of dividing the B-value by the A-value (B-value / A-value) and queues the next instruction (PC + 1). DIV.I functions as DIV.F would. If the A-value is zero, the B-value is unchanged and the current task is removed from the warrior's task queue.

8.5.8 MOD

MOD replaces the B-target with the integral remainder of dividing the B-value by the A-value (B-value % A-value) and queues the next instruction (PC + 1). MOD.I functions as MOD.F would. If the A-value is zero, the B-value is unchanged and the current task is removed from the warrior's task queue.

8.5.9 JMP

MP queues the sum of the program counter and the A-pointer.

8.5.10 JMZ

JMZ tests the B-value to determine if it is zero. If the B-value is zero, the sum of the program counter and the A-pointer is queued. Otherwise, the next instruction is queued (PC + 1). JMZ.I functions as JMZ.F would, i.e. it jumps if both the A-number and the B-number of the B-instruction are zero.

8.5.11 JMN

JMN tests the B-value to determine if it is zero. If the B-value is not zero, the sum of the program counter and the A-pointer is queued. Otherwise, the next instruction is queued (PC + 1). JMN.I functions as JMN.F would, i.e. it jumps if both the A-number and the B-number of the B-instruction are non-zero. This is not the negation of the condition for JMZ.F.

8.5.12 DJN

DJN decrements the B-value and the B-target, then tests the B-value to determine if it is zero. If the decremented B-value is not zero, the sum of the program counter and the A-pointer is queued. Otherwise, the next instruction is queued (PC + 1). DJN.I functions as DJN.F would, i.e. it decrements both both A/B-numbers of the B-value and the B-target, and jumps if both A/B-numbers of the B-value are non-zero.

8.5.13 CMP

CMP compares the A-value to the B-value. If the result of the comparison is equal, the instruction after the next instruction (PC + 2) is queued (skipping the next instruction). Otherwise, the the next instruction is queued (PC + 1).

8.5.14 SLT

SLT compares the A-value to the B-value. If the A-value is less than the B-value, the instruction after the next instruction (PC + 2) is queued (skipping the next instruction). Otherwise, the next instruction is queued (PC + 1). SLT.I functions as SLT.F would.

8.5.15 SPL

SPL queues the next instruction (PC + 1) and then queues the sum of the program counter and A-pointer. If the queue is full, only the next instruction is queued.

8.5.16 CTIN

CTIN queues the next instruction (PC + 1). It stores the incoming values of the intra-process communication in both A-field and B-field. Then the data can be accessed as the DAT instruction. However its execution doesn't kill the process.

8.5.17 CTOUT

CTOUT queues the next instruction (PC + 1). It send both values in the A-field and B-field to the intra-process communication channel. All address modes are valid.

8.5.18 CPIN

CPIN queues the next instruction (PC + 1). It stores the incoming values of the inter-process communication in both A-field and B-field. Then the data can be accessed as the DAT instruction. However its execution doesn't kill the process.

8.5.19 CPOUT

CPOUT queues the next instruction (PC + 1). It send both values in the A-field and B-field to the inter-process communication channel. All address modes are valid.

8.6 Address Modes

The WiCE language has 8 addressing modes:

- # – immediate
- $ – direct (the $ may be omitted)
- * – A-field indirect
- @ – B-field indirect
- { – A-field indirect with predecrement
- < – B-field indirect with predecrement
- } – A-field indirect with postincrement
- > – B-field indirect with postincrement

One important thing to remember about the predecrement and postincrement modes is that the pointers will be in-/decremented even if they're not used for anything. So JMP -1, <100 would decrement the instruction 100 even if the value it points to isn't used for anything. Even DAT <50, <60 will decrement the addresses in addition to killing the process.

8.6.1 Immediate

An immediate mode operand merely serves as storage for data. An immediate A/B-mode in the current instruction sets the A/B-pointer to zero.

8.6.2 Direct

A direct mode operand indicates the offset from the program counter. A direct A/B-mode in the current instruction means the A/B-pointer is a copy of the offset, the A/B-number of the current instruction.

8.6.3 Indirect

An indirect mode operand indicates the primary offset (relative to the program counter) to the secondary offset (relative to the location of the instruction in which the secondary offset is contained). An indirect A/B-mode indicates that the A/B-pointer is the sum of the A/B-number of the current instruction (the primary offset) and the B-number of the instruction pointed to by the A/B-number of the current instruction (the secondary offset).

8.6.4 Predecrement Indirect

A predecrement indirect mode operand indicates the primary offset (relative to the program counter) to the secondary offset (relative to the location of the instruction in which the secondary offset is contained) which is decremented prior to use. A predecrement indirect A/B-mode indicates that the A/B-pointer is the sum of the A/B-number of the current instruction (the primary offset) and the decremented B-number of the instruction pointed to by the A/B-number of the current instruction (the secondary offset).

8.6.5 Postincrement Indirect

A postincrement indirect mode operand indicates the primary offset (relative to the program counter) to the secondary offset (relative to the location of the instruction in which the secondary offset is contained) which is incremented after the results of the operand evaluation are stored. A postincrement indirect A/B-mode indicates that the A/B-pointer is the sum of the A/B-number of the current instruction (the primary offset) and the B-number of the instruction pointed to by the A/B-number of the current instruction (the secondary offset). The B-number of the instruction pointed to by the A/B-number of the current instruction is incremented after the A/B-instruction is stored, but before the B-operand is evaluated (for post-increment A-mode), or the operation is executed (for post-increment indirect B-mode).

8.7 Modifiers

The modifiers are suffixes that are added to the instruction to specify which parts of the source and the destination it will affect. For example, MOV.AB 4, 5 would move the A-field of the instruction 4 into the B-field of the instruction 5. There are 7 different modifiers available:

- MOV.A – moves the A-field of the source into the A-field of the destination
- MOV.B – moves the B-field of the source into the B-field of the destination
- MOV.AB – moves the A-field of the source into the B-field of the destination
- MOV.BA – moves the B-field of the source into the A-field of the destination
- MOV.F – moves both fields of the source into the same fields in the destination
- MOV.X – moves both fields of the source into the opposite fields in the destination
- MOV.I – moves the whole source instruction into the destination

Naturally the same modifiers can be used for all instructions, not just for MOV. Some instructions like JMP and SPL, however, don't care about the modifiers.
Since not all the modifiers make sense for all the instructions, they will default to the closest one that does make sense. The most common case involves the .I modifier: To keep the language simple and abstract no numerical equivalents have been defined for the OpCodes, so using mathematical operations on them wouldn't make any sense at all. This means that for all instructions except MOV, SEQ and SNE (and CMP which is just an alias for SEQ) the .I modifier will mean the same as the .F.
When modifiers are omitted, WiCE will put default modifiers (Table 8.2).

8.7.1 A

Instruction execution proceeds with the A-value set to the A-number of the A-instruction and the B-value set to the A-number of the B-instruction. A write to core replaces the A-number of the instruction pointed to by the B-pointer.
For example, a CMP.A instruction would compare the A-number of the A-instruction with the A-number of the B-instruction. A MOV.A instruction would replace the A-number of the instruction pointed to by the B-pointer with the A-number of the A-instruction.

Instruction	**Default Modifier**
DAT, NOP	Always .F, but it's ignored
MOV, SEQ, SNE, CMP	If A-mode is immediate, .AB, if B-mode is immediate and A-mode isn't, .B, if neither mode is immediate, .I.
ADD, SUB, MUL, DIV, MOD	If A-mode is immediate, .AB, if B-mode is immediate and A-mode isn't, .B, if neither mode is immediate, .F.
SLT, LDP, STP	If A-mode is immediate, .AB, if it isn't, (always!) .B.
JMP, JMZ, JMN, DJN, SPL	Always .B (but it's ignored for JMP and SPL).

Table 8.2: The Default Modifiers when Omitted

8.7.2 B

Instruction execution proceeds with the A-value set to the B-number of the A-instruction and the B-value set to the B-number of the B-instruction. A write to core replaces the B-number of the instruction pointed to by the B-pointer.
For example, a CMP.B instruction would compare the B-number of the A-instruction with the B-number of the B-instruction. A MOV.B instruction would replace the B-number of the instruction pointed to by the B-pointer with the B-number of the A-instruction.

8.7.3 AB

Instruction execution proceeds with the A-value set to the A-number of the A-instruction and the B-value set to the B-number of the B-instruction. A write to core replaces the B-number of the instruction pointed to by the B-pointer.
For example, a CMP.AB instruction would compare the A-number of the A-instruction with the B-number of the B-instruction. A MOV.AB instruction would replace the B-number of the instruction pointed to by the B-pointer with the A-number of the A-instruction.

8.7.4 BA

Instruction execution proceeds with the A-value set to the B-number of the A-instruction and the B-value set to the A-number of the B-instruction. A write to core replaces the A-number of the instruction pointed to by the B-pointer.
For example, a CMP.BA instruction would compare the B-number of the A-instruction with the A-number of the B-instruction. A MOV.BA instruction would replace the A-number of the instruction pointed to by the B-pointer with the B-number of the

A-instruction.

8.7.5 F

Instruction execution proceeds with the A-value set to both the A-number and B-number of the A-instruction (in that order) and the B-value set to both the A-number and B-number of the B-instruction (also in that order). A write to core replaces both the A-number and the B-number of the instruction pointed to by the B-pointer (in that order).
For example, a CMP.F instruction would compare the A-number of the A-instruction to the A-number of the B-instruction and the B-number of the A-instruction to B-number of the B-instruction. A MOV.F instruction would replace the A-number of the instruction pointed to by the B-pointer with the A-number of the A-instruction and would also replace the B-number of the instruction pointed to by the B-pointer with the B-number of the A-instruction.

8.7.6 X

Instruction execution proceeds with the A-value set to both the A-number and B-number of the A-instruction (in that order) and the B-value set to both the B-number and A-number of the B-instruction (in that order). A write to to core replaces both the B-number and the A-number of the instruction pointed to by the B-pointer (in that order).
For example, a CMP.X instruction would compare the A-number of the A-instruction to the B-number of the B-instruction and the B-number of the A-instruction to A-number of the B-instruction. A MOV.X instruction would replace the B-number of the instruction pointed to by the B-pointer with the A-number of the A-instruction and would also replace the A-number of the instruction pointed to by the B-pointer with the B-number of the A-instruction.

8.7.7 I

Instruction execution proceeds with the A-value set to the A-instruction and the B-value set to the B-instruction. A write to core replaces the entire instruction pointed to by the B-pointer.
For example, a CMP.I instruction would compare the A-instruction to the B-instruction. A MOV.I instruction would replace the instruction pointed to by the B-pointer with the A-instruction.

Chapter 9

The WiCE environment

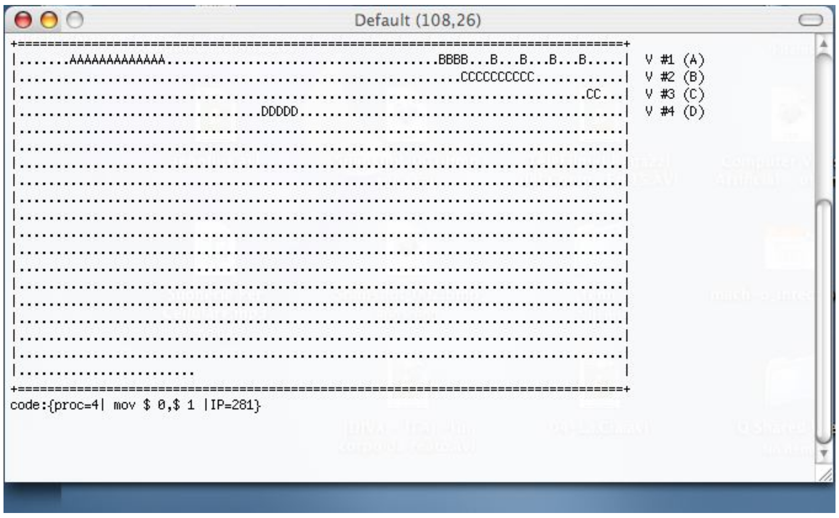
9.1 Using WiCE

To compile the program just type *make* from the source directory.
To run the program just type the *wice* command with the proper command line options. WiCE is a very flexible viral code simulator, and has different options:

```
$ ./wice

WiCE v0.1 by Fernando Iazeolla

USAGE:
wice [options] file1 file2 file3 ...

valid options:

--file(-f) file     alternative log file
--output(-o) output output mode (quiet,normal,debug,debug2,debug3)
--size(-m) size     size of the array
--comm(-c) comm     communication type(null,process,thread)
--vo(-v) mode       modes are (none,txt,x11)
--log(-l)           turn log on (off default)
--sleeptime(-z)     pause between step commands (in seconds)
--gtkwaittime(-w)   pause between step commands (in milliseconds)
--multiwin(-e)      toggle multi or single window(s) (default is multi windows)
--cpu=cicles(-c)    set the cpu cicles. -1 means infinite untill there is a winner.
```

By default WiCE writes the essential informations on the standard output. You can redirect the output on a log file with the -l option. You can also control the log file name via the -f option.
There are different level of output details. You can control this via the -o option.
On the default mode WiCE doesn't offers any GUI. You can set the preferred GUI by setting the appropriate value on the –vo option. The *txt* value generates an ascii

GUI (Figure 9.1). The *x11* value generates a windowed output. By default the x11 GUI has separate windows (Figure 9.2), but one can set a unique window with the -e option (Figure 9.3). The GUI level is completely transparent to other WiCE architectural under-level. The -c option controls the communication level between agents.



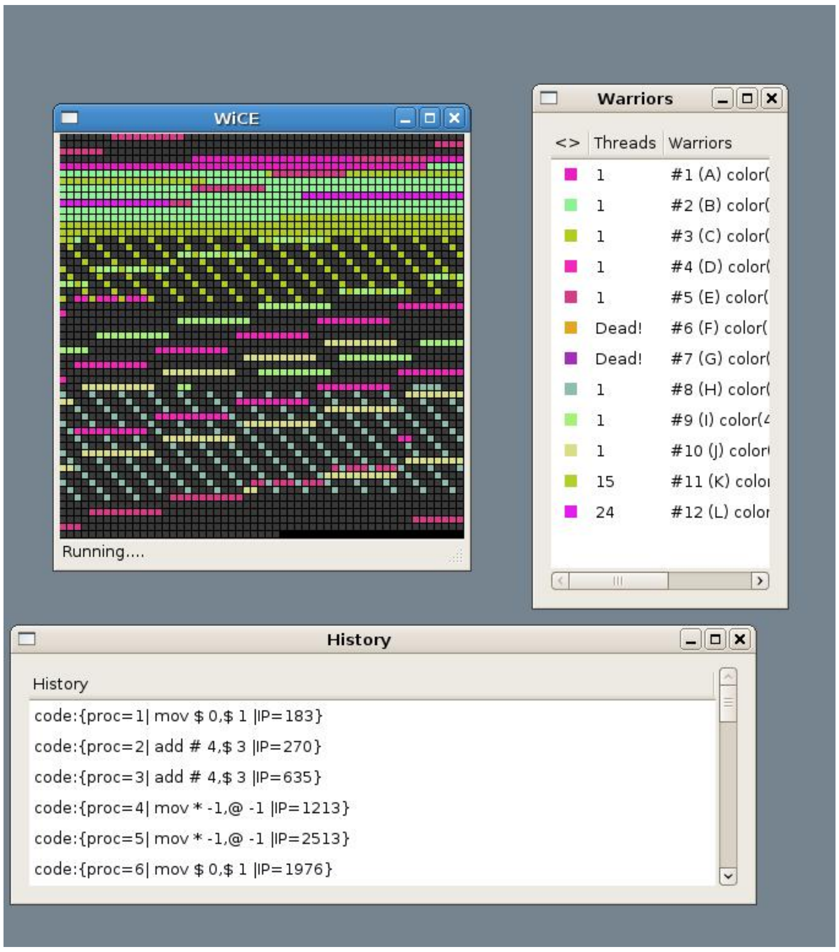
```
+================================================================================+
|.......AAAAAAAAAAAAA...................................BBBB...B...B...B...B.....|  V #1 (A)
|..........................................................CCCCCCCCCC............|  V #2 (B)
|...........................................................................CC...|  V #3 (C)
|................................DDDDD...........................................|  V #4 (D)
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|................................................................................|
|.......................                                                         |
+================================================================================+
code:{proc=4| mov $ 0,$ 1 |IP=281}
```

Figure 9.1: The txt mode GUI of WiCE

The *null* option inhibits all communication type from and to any process or thread. The *process* option allow only communication between different processes while the *thread* option allow only communication between threads of the same process.
The wait time between single commands can be controlled via the *sleeptime* or the *gtkwaittime* options depending on which video output (vo) has been selected. This because the windowed ambient has an event-orientation programming, so the execution routine has been implemented in a different way from the classical sequential programming method.
The –cpu option can be used to control the exit condition of WiCE depending on the actual CPU cycle.
The –size option sets the memory array's dimension.

9.2 WiCE Description

The WiCE environment is able to run two or more self-modifying codes in a "sandbox" that separates the viral codes from the real machine.
The language interpreted by WiCE is machine independent. This means that the programmer can focus on optimizing code and self-modifying/repairing techniques without worrying about the specific machine layer.

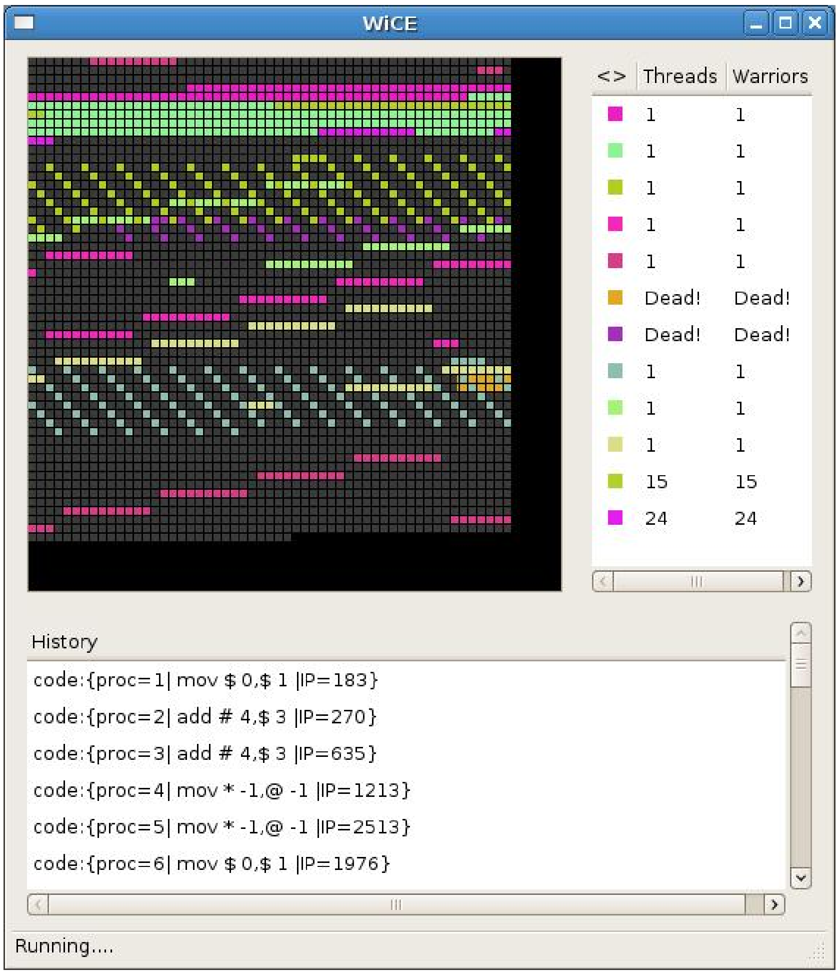


<>	Threads	Warriors
🟪	1	#1 (A) color(
🟩	1	#2 (B) color(
🟩	1	#3 (C) color(
🟪	1	#4 (D) color(
🟥	1	#5 (E) color(
🟧	Dead!	#6 (F) color(
🟪	Dead!	#7 (G) color(
🟩	1	#8 (H) color(
🟩	1	#9 (I) color(4
🟨	1	#10 (J) color
🟩	15	#11 (K) colo
🟪	24	#12 (L) color

```
History
code:{proc=1| mov $ 0,$ 1 |IP=183}
code:{proc=2| add # 4,$ 3 |IP=270}
code:{proc=3| add # 4,$ 3 |IP=635}
code:{proc=4| mov * -1,@ -1 |IP=1213}
code:{proc=5| mov * -1,@ -1 |IP=2513}
code:{proc=6| mov $ 0,$ 1 |IP=1976}
```

Figure 9.2: The multi window x11 GUI of WiCE

WiCE offers a secure environment to develop and test viral code and techniques without smash any real computer.
The WiCE environments also offers inter/intra viral-process communication APIs. This opens to a very interesting new scenario: the game theory.



<>	Threads	Warriors
🟪	1	1
🟩	1	1
🟨	1	1
🟪	1	1
🟥	1	1
🟧	Dead!	Dead!
🟪	Dead!	Dead!
🟦	1	1
🟩	1	1
🟨	1	1
🟨	15	15
🟪	24	24

History

```
code:{proc=1| mov $ 0,$ 1 |IP=183}
code:{proc=2| add # 4,$ 3 |IP=270}
code:{proc=3| add # 4,$ 3 |IP=635}
code:{proc=4| mov * -1,@ -1 |IP=1213}
code:{proc=5| mov * -1,@ -1 |IP=2513}
code:{proc=6| mov $ 0,$ 1 |IP=1976}
```

Running....

Figure 9.3: The one window x11 GUI of WiCE

Chapter 10

WiCE Internals

10.1 Software Architecture

WiCE is a flexible simulating environment. The language interpreted by WiCE is an evolution of the ICWS'94 (the Redcode standard of 1994). This language is a high level assembler that is independent from the underlying layer. So the programmer doesn't have to worry about the machine's hardware architecture and can focus on important matters such as self-replication and self-modifying algorithms.
The WiCE framework can be divied into four main logical layers (Figure 10.1): the input&compiling layer, the initialization layer, the runtime layer, and the output layer. Each of them is divided again in multiple sub-layers. Each layer has a specific issue and can be changed or completely substituted without interfering with the other layers.
In the next section we will describe all of the layers of the WiCE framework.

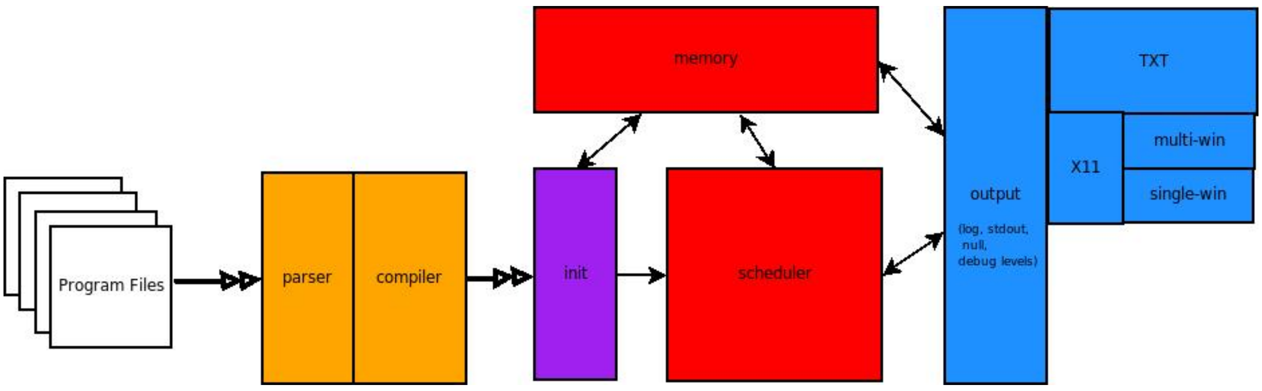


Figure 10.1: The WiCE Software Architecture

10.2 the parser

The parser processes the input files in two stages. During the first stage, the stream is tokenized and the grammar is verified syntactically. On the second stage semantic verification and address resolving is performed.
So in the first stage we first initialize the main process structure (Figure 10.2): and

```
struct Process{
 int processID;
 struct process_task *pt;
 struct process_construct *pc;
 struct Process *prev,*next;
};
```

Figure 10.2: The main Process structure

we are going to fill the "process_construct" structure (Figure 10.3). This structure is used while compiling the source viral program. In this structure the "org" field is

```
struct process_construct{
 struct instruction_node *first,*last;
 int len;
 char org[MAXSTR];
 struct var_table *vt_first,*vt_last;
};
```

Figure 10.3: The process_construct structure

the label of the entry point of the viral code. In this first stage of parsing process, only the label name is detected, and then it will be resolved into a valid address into the next parsing stage.
The struct "var_table" represent the variable table associated to this process. In this table there is the variable names and the relative expressions is its expanded mode, not already solved. Perhaps if we have a source line like:

```
x = 3 + 1 * 5 ;
```

in the variable table the name would be equal to "x", and the expression will point to a structure that represent: "$+(3,*(1,5))$".
The expression will be solved during the next stage. This because if in the right side of the equation there would be an other variable, this might be solved also into a valid number into the next stage.
The instruction node structure (Figure 10.4) is the core structure of the first stage

```
struct instruction_node{
 char instr [MAXSTR];
 char modifier[6];
 char laddr[4];
 char raddr[4];
 int num_node;
 int line_count;
 struct expr_node *left,*right;
 void *code;
 struct instruction_node *prev,*next;
};
```

Figure 10.4: The Instruction Node Structure

parsing process. For each instruction a new instruction node will be filled. During this parsing stage the token are verified but are not converted into opcodes. So a text field is passed to each of the instruction node fields. the "laddr" and "raddr" fields represent the left and right addressing mode respectively. The "line_count" field is filled with source line number related to the instruction that is being processing. This helps for debugging purpose, so the line number relatively to an eventually error can be printed. The "num_code" is the relative address from the beginning of the code. This is used during the second parsing stage to solve code addresses. The "num_code" and the "line_count" can be different because more than one instruction can be inserted in one line or comments line can be present in the source code and they are not counted in instruction nodes.
The expression node (Figure 10.5), is a structure where variable expressions are stored before being solved in the second stage. In the "str" field is stored the expression sym-

```
struct expr_node{
 int type;
 char str[MAXSTR];
 struct expr_node *left,*right;
};
```

Figure 10.5: The Expression Node

bol. The type field identity the symbol and can assume the values listed in Figure 10.6. The precedence order is given by : $OP\_BOOL > OP\_MUL > OP\_ADD$. The "code" field in the instruction node structure (Fig.10.4) is for the moment unused. It will be used in the second parsing stage and it will be the complete opcode of the instruction that will be fetched into the memory.
The first parser stage ends by adding into the process's variable space all the WiCE

```
#define OP_NULL 0
#define OP_MUL 1
#define OP_ADD 2
#define NUMBER 3
#define VAR 4
#define OP_BOOL 5
```

Figure 10.6: Types of expression items

environment constants such as CORESIZE, WARRIORS, MAXPROCESSES, MAXCYCLES, MAXLENGTH, MINDISTANCE, and VERSION.

10.3 the compiler

The compiler (or second parsing) stage starts by solving all the assertions and immediately stops if one of them returns a false condition.
Then all instruction node fields are converted into a coded binary format and the "unpacked_op_mem" (Figure 10.7) is filled. This structure is used even in decom-

```
struct unpacked_op_mem{
 unsigned int processID;
 int opcode;
 int mod;
 int a_pref;
 int b_pref;
 int a_val;
 int b_val;
};
```

Figure 10.7: Unpacked Op Mem Structure

piling opcodes from memory to perform the relative action easily. Once the "unpacked_op_mem" is compiled properly the "pack" function creates the entire opcode and puts it into the "code" field of the instruction node structure. This field is a "void" pointer. This because the WiCE frameworks has different memory models, and actually an opcode can be represented by a 32-bit, 64-bit- or a 128-bit words (Figure 10.8). So the "code" field is initialized at run time. New memory alignment can be inserted in the future.
During this second parsing stage, all variables value and memory address are solved. Even the program's entry point.

`32-bit:`	`process_id`	`(31-28)`	`4-bits`
	`opcode`	`(27-23)`	`5-bits`
	`modifier`	`(22-19)`	`4-bits`
	`a_field_addr_mode`	`(18-16)`	`3-bits`
	`b_field_addr_mode`	`(15-13)`	`3-bits`
	`a_value`	`(12-7)`	`6-bits`
	`b_value`	`(6-1)`	`6-bits`
	`unused`	`(0)`	`1-bit`
`64-bit:`	`process_id`	`(31-16)`	`16-bits`
	`opcode`	`(15-12)`	`4-bits`
	`modifier`	`(11-8)`	`4-bits`
	`a_field_addr_mode`	`(7-5)`	`3-bits`
	`b_field_addr_mode`	`(4-2)`	`3-bits`
	`unused`	`(1-0)`	`2-bits`
	`a_value`	`(31-16)`	`16-bits`
	`b_value`	`(15-0)`	`16-bits`
`128-bit:`	`process_id`	`(31-0)`	`32-bits`
	`opcode`	`(31-28)`	`4-bits`
	`modifier`	`(27-25)`	`3-bits`
	`a_field_addr_mode`	`(24-22)`	`3-bits`
	`b_field_addr_mode`	`(21-0)`	`22-bits padded`
	`a_value`	`(31-0)`	`32-bits`
	`b_value`	`(31-0)`	`32-bits`

Figure 10.8: WiCE Memory Models

10.4 the initializer

The initializer starts by allocating the memory according to the memory model (Figure 10.8) and the memory size specified in the command line.
Then, for all the processes, it puts the process's code, located in the code(s) field of the process construct, into the memory at the right position.
Then the process task structure (Figure 10.9) and its only initial thread (Figure 10.10) are initialized. The memory related to the "process_construct" is no more needed and released.
Then, according to the video output selected at the command line options, the proper video output initializer is called (none, txt, x11). The video output initializer sets the memory arena output, the warrior list and the history space.

```
struct process_task{
 unsigned int ID;
 unsigned int n_threads;
 struct process_thread *cur_thread,*primo_thread,*ultimo_thread;
 int communication_in_a,communication_in_b,communication_out_a,communication_out
 char out_symbol;
 int out_color;
 GdkColor m_color;
 struct process_task *prev,*next;
};
```

Figure 10.9: The Process Task Structure

```
struct process_thread{
 unsigned int IP;
 struct process_task *ptask;
 int communication_in_a,communication_in_b,communication_out_a,communication_out
 struct process_thread *prev,*next;
};
```

Figure 10.10: The Process Thread Structure

10.5 the scheduler

The scheduler routine takes the first waiting thread of the first process waiting on the processes queue. Then it unpacks the current command and executes it. If the process dies then it delete the process from the main process queue. The pointer to the next process on the main list is updated, as also the next thread on the process list if the process executed is still alive.
The loop ends if either there is only one process alive or the maximum CPU cycle is reached. In the second case the match is considered a draw.
In the x11 video output mode there is another scheduler function. This because in this case the function is called by a timeout event, because of the event-driven model used under x11 mode. Anyway the structure of both scheduler functions are very similar. They differ only for the x11 widgets refresh system that is a consequence of the x11 event-driven model instead of the modular textual programming mode that uses a different interaction mode.

10.6 the output

There are different output layers as depicted in Figure 10.1. The "none" output mode does not produce any output on the standard output stream. The main output layer includes debug output streams that can be printed in the standard output as default or in a log file. If you decide to have a graphical output, there is a text semi-graphical output (Figure 9.1) and a X11 GUI graphical (Figure 9.2) layer above the main output.

Appendix A

WiCE SourceCode

The WiCE source code has been released under GPLv2 licence (GNU Public Licence version 2). The actual repository is located at **http://code.google.com/p/wice/**
.
The snapshot of the source code on which this thesis is based is listed below.

A.1 main.h

```
#define OUTPUT_QUIET 1
#define OUTPUT_NORMAL 2
#define OUTPUT_DEBUG 3
#define OUTPUT_DEBUG2 4
#define OUTPUT_DEBUG3 5
#define VO_NONE 1
#define VO_FRAMEBUFFER 2
#define VO_X11 3

#define ARENA_SIZE 8000

#define COMM_NULL 0
#define COMM_PROC 1
#define COMM_THREAD 2

#define MAX_CPU_CICLE 10000
#define STAT_WAIT_TIME 1000

#define MEM_TYPE_ONE 1
#define MEM_TYPE_TWO 2
#define MEM_TYPE_FOUR 4

#define MAXSTR 255
#define MAXMOD 6
```

```
#define MAX_PROG_SIZE 200
#define MAXPROCESSES 1000
#define MINDISTANCE 200
#define VERSION 01
#define PSPACESIZE 500

#define VERBOSE 1
#define DO_DEBUG 1

#define OP_NULL 0
#define OP_MUL 1
#define OP_ADD 2
#define NUMBER 3
#define VAR 4
#define OP_BOOL 5
#define ASSERT_STR "ASSERT"
#define ALIVE 1
#define DEAD 0

struct array_mem_small{
unsigned int mem;
};
struct array_mem_mid{
unsigned int processID_opcode;
unsigned int arg1_arg2;
};
struct array_mem_norm{
unsigned int processID;
unsigned int opcode;
unsigned int arg1;
unsigned int arg2;
};
struct expr_node{
int type;
char str[MAXSTR];
struct expr_node *left,*right;
};
struct instruction_node{
char instr [MAXSTR];
char modifier[6];
char laddr[4];
char raddr[4];
int num_node;
int line_count;
struct expr_node *left,*right;
void *code;
struct instruction_node *prev,*next;
};
struct process_task{
unsigned int ID;
unsigned int n_threads;
struct process_thread *cur_thread,*primo_thread,*ultimo_thread;
int communication_in_a,communication_in_b,communication_out_a,communication_out_b;
char out_symbol;
int out_color;
GdkColor m_color;
struct process_task *prev,*next;
};
struct process_thread{
unsigned int IP;
struct process_task *ptask;
int communication_in_a,communication_in_b,communication_out_a,communication_out_b;
struct process_thread *prev,*next;
};
struct var_table{
char name[MAXSTR];
struct expr_node *val_first,*val_last;
struct var_table *prev,*next;
};
struct process_construct{
struct instruction_node *first,*last;
int len;
char org[MAXSTR];
struct var_table *vt_first,*vt_last;
};
struct Process{
int processID;
struct process_task *pt;
struct process_construct *pc;
struct Process *prev,*next;
};
struct unpacked_op_mem{
unsigned int processID;
int opcode;
int mod;
int a_pref;
int b_pref;
int a_val;
int b_val;
};
```

```
void *arena;
int size_arena,warriors,version,min_distance,maxprocesses,max_prog_size;
int arena_mem_type;
int output_mode,vo_mode,log_mode,b_log,max_x,max_y,sc_x,sc_y;
int xd,yd,xl,yl,max_sc_y,sleeptime,b_multi_win,gtkwaittime,current_make_node,
txt_y_warrior_list,g_actual_CPU;
int gtk_sc_x,gtk_sc_y;
int communication;
int CPU_cicle;
struct process_task *primo_task,*ultimo_task,*first_killed_task,*last_killed_task;
struct Process *proc_primo,*proc_ultimo;
char logfile[MAXSTR],out_str[MAXSTR];
FILE *fpout;
```

A.2 main.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<getopt.h>
#include<string.h>
#include<errno.h>
#include<gtk/gtk.h>
#include"main.h"

void usage()
{
printf("WiCE v0.1 by xnando\n");
printf("\nUSAGE:\n");
printf("wice [options] file1 file2 file3 ...\n");
printf("\nvalid options:\n\n");
printf("\t--file(-f) file\t\talternative log file\n");
printf("\t--output(-o) output\toutput mode
(quiet,normal,debug,debug2,debug3)\n");
printf("\t--size(-m) size\t\tsize of the array\n");
printf("\t--comm(-c) comm\t\tcommunication type(null,process,thread)\n");
printf("\t--vo(-v) mode\t\tmodes are (none,txt,x11)\n");
printf("\t--log(-l)\t\tturn log on (off default)\n");
printf("\t--sleeptime(-z)\t\tpause between step commands (in seconds)\n");
printf("\t--gtkwaittime(-w)\tpause between step commands (in
milliseconds)\n");
printf("\t--multiwin(-e)\t\ttoggle multi or single window(s) (default is
multi windows)\n");
printf("\t--cpu=cicles(-c)\tset the cpu cicles. -1 means infinite untill
there is a winner.\n");
exit(1);
}
void get_defaults_arg()
{
size_arena=ARENA_SIZE;
CPU_cicle=MAX_CPU_CICLE;
arena_mem_type=MEM_TYPE_FOUR;
output_mode=OUTPUT_NORMAL;
communication=COMM_NULL;
warriors=0;
version=VERSION;
min_distance=MINDISTANCE;
maxprocesses=MAXPROCESSES;
max_prog_size=MAX_PROG_SIZE;
strcpy(logfile,"wice.log");
log_mode=0;
vo_mode=VO_NONE;
fpout=stdout;
b_log=0;
sleeptime=0;
b_multi_win=1;
gtkwaittime=5;
gtk_sc_x=200;
gtk_sc_y=200;
}
void parse_args(int argc,char **argv)
{
int c;
while (1)
{
static struct option long_options[] =
{
{"output",required_argument,0,'o'},
{"memtype",required_argument,0,'m'},
{"cpu",required_argument,0,'c'},
{"comm",required_argument,0,'z'},
{"file",required_argument,0,'f'},
{"size",required_argument,0,'s'},
{"vo",required_argument,0,'v'},
{"sleeptime",required_argument,0,'z'},
{"gtkwaittime",required_argument,0,'w'}
};
int option_index = 0;
c = getopt_long (argc, argv, "hlev:o:m:c:z:f:k:w:",long_options,
&option_index);
if (c == -1) break;
switch(c)
{
case 'o':
if((strcmp(optarg,"quiet"))==0) output_mode=OUTPUT_QUIET;
if((strcmp(optarg,"normal"))==0) output_mode=OUTPUT_NORMAL;
if((strcmp(optarg,"debug"))==0) output_mode=OUTPUT_DEBUG;
if((strcmp(optarg,"debug2"))==0) output_mode=OUTPUT_DEBUG2;
if((strcmp(optarg,"debug3"))==0) output_mode=OUTPUT_DEBUG3;
break;
case 'm':
if((strcmp(optarg,"tiny"))==0) arena_mem_type=MEM_TYPE_ONE;
if((strcmp(optarg,"medium"))==0) arena_mem_type=MEM_TYPE_TWO;
if((strcmp(optarg,"large"))==0) arena_mem_type=MEM_TYPE_FOUR;
break;
case 'c':
CPU_cicle=strtol(optarg,NULL,10);
if(errno==EINVAL) usage();
break;
case 'k':
if((strcmp(optarg,"null"))==0) communication=COMM_NULL;
if((strcmp(optarg,"process"))==0) communication=COMM_PROC;
if((strcmp(optarg,"thread"))==0) communication=COMM_THREAD;
break;
case 'f':
strncpy(logfile,optarg,MAXSTR);
break;
case 's':
size_arena=strtol(optarg,NULL,10);
if(errno==EINVAL) usage();
break;
case 'z':
sleeptime=strtol(optarg,NULL,10);
if(errno==EINVAL) usage();
break;
case 'w':
gtkwaittime=strtol(optarg,NULL,10);
if(errno==EINVAL) usage();
break;
case 'v':
if((strcmp(optarg,"none"))==0) vo_mode=VO_NONE;
if((strcmp(optarg,"txt"))==0) vo_mode=VO_FRAMEBUFFER;
if((strcmp(optarg,"x11"))==0) vo_mode=VO_X11;
break;
```

```
case 'l':
b_log=1;log_mode=1;
break;
case 'e':
b_multi_win=0;
break;
case 'h':
case '?':
usage();
break;
default:
usage();
break;
}

}
}
void look_data_ok()
{
if(arena_mem_type==MEM_TYPE_ONE)
{
if(size_arena>64) die("in tiny model the size of the arena must be<64 or
try a bigger model");
if(max_prog_size>64) die("in tiny model the size of the warrior must
be<64 or try a bigger model");
if(maxprocesses>16) die("in tiny model maximum nember of processes must
be<16 or try a bigger model");

}
if(arena_mem_type==MEM_TYPE_TWO)
{
if(size_arena>65535) die("in tiny model the size of the arena must
be<65535 or try a bigger model");
if(max_prog_size>65535) die("in tiny model the size of the warrior must
be<65535 or try a bigger model");
if(maxprocesses>65535) die("in tiny model maximum nember of processes
must be<65535 or try a bigger model");
}
if(arena_mem_type==MEM_TYPE_FOUR)
{

}
}
int main(int argc,char **argv)
{
```

```
proc_primo=NULL;
proc_ultimo=NULL;
primo_task=NULL;
ultimo_task=NULL;
get_defaults_arg();
parse_args(argc,argv);
if(b_log)
{
fpout=fopen(logfile,"w");
if(fpout==NULL)
{
printf("error opening log file\n");
exit(1);
}
}
look_data_ok();
if(output_mode>=OUTPUT_DEBUG)
{
sprintf(out_str,"outputmode=%d\n",output_mode);
fputs(out_str,fpout);
sprintf(out_str,"memsize=%d\n",size_arena);
fputs(out_str,fpout);
sprintf(out_str,"b_log=%d\n",b_log);
fputs(out_str,fpout);
sprintf(out_str,"vo=%d\n",vo_mode);
fputs(out_str,fpout);
sprintf(out_str,"sleeptime=%d\n",sleeptime);
fputs(out_str,fpout);
}
//read_files (and parse it in 2 passes
warriors=argc-optind;
if(optind<argc)
{
while(optind<argc)
{
if(output_mode>=OUTPUT_NORMAL)
{
sprintf(out_str,"parsing %s ... ",argv[optind]);
fputs(out_str,fpout);
}
parse(argv[optind++]);
if(output_mode>=OUTPUT_NORMAL)
{
sprintf(out_str,"ok\n");
fputs(out_str,fpout);
}
}
}
else
{
usage();
}
//init_game
init_game();
//play_game
play_game();
//result

deinit_game();
if(b_log) fclose(fpout);
return 0;
}
```

A.3 init_game.h

```
void init_game(void);
void deinit_game(void);
```

A.4 init_game.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<strings.h>
#include<errno.h>
#include<ctype.h>
#include<math.h>
#include<gtk/gtk.h>
#include"main.h"
#include"list_utils.h"
#include"parse2.h"
#include"pack.h"
#include"init_game.h"
#include"txt_output.h"
#include"x11_output.h"

extern GdkColormap *execute_colormap;
```

```
void putcode(int pos,int processID,void *code)
{
struct array_mem_small *msmall,*small_arena;
struct array_mem_mid *mmid,*mid_arena;
struct array_mem_norm *mlarge,*large_arena;
if(arena_mem_type==MEM_TYPE_ONE)
{
msmall=(struct array_mem_small*)code;
small_arena=(struct array_mem_small*)arena;
small_arena[pos].mem=msmall->mem;
}
if(arena_mem_type==MEM_TYPE_TWO)
{
mmid=(struct array_mem_mid*)code;
mid_arena=(struct array_mem_mid*)arena;
mid_arena[pos].processID_opcode=mmid->processID_opcode;
mid_arena[pos].arg1_arg2=mmid->arg1_arg2;
}
if(arena_mem_type==MEM_TYPE_FOUR)
{
mlarge=(struct array_mem_norm*)code;
large_arena=(struct array_mem_norm*)arena;
large_arena[pos].processID=mlarge->processID;
large_arena[pos].opcode=mlarge->opcode;
large_arena[pos].arg1=mlarge->arg1;
large_arena[pos].arg2=mlarge->arg2;
}
}
void get_symbols(char *out_symbol,int *out_color)
{
static color=1;
static letter='A';

*out_symbol=letter++;
*out_color=color++;
}
void free_expr(struct expr_node *expr)
{
if(expr->left) free_expr(expr->left);
if(expr->right) free_expr(expr->right);
free(expr);
}
void init_game()
{
struct array_mem_small *msmall;
struct array_mem_mid *mmid;
struct array_mem_norm *mlarge;
struct Process *proc;
struct instruction_node *in;
struct process_thread *pthread;
struct process_task *ptask;
int next_offs,proc_offs,pos;
double xx,lato_x,lato_y;
if(arena_mem_type==MEM_TYPE_ONE)
{
msmall=(struct array_mem_small*)malloc(sizeof(struct array_mem_small)*size_arena);
if(msmall==NULL) die("error malloking small array mem");
bzero(msmall,sizeof(struct array_mem_small)*size_arena);
arena=msmall;
}
if(arena_mem_type==MEM_TYPE_TWO)
{
mmid=(struct array_mem_mid*)malloc(sizeof(struct array_mem_mid)*size_arena);
if(mmid==NULL) die("error malloking mid array mem");
bzero(mmid,sizeof(struct array_mem_mid)*size_arena);
arena=mmid;
}
if(arena_mem_type==MEM_TYPE_FOUR)
{
mlarge=(struct array_mem_norm*)malloc(sizeof(struct array_mem_norm)*size_arena);
if(mlarge==NULL) die("error malloking large array mem");
bzero(mlarge,sizeof(struct array_mem_norm)*size_arena);
arena=mlarge;
}
next_offs=0;
for(proc=proc_primo;proc;proc=proc->next)
{
//calc offset in mem
proc_offs=(next_offs+(rand()%min_distance))%size_arena;
//put in mem
pos=proc_offs;
for(in=proc->pc->first;in;in=in->next)
{
putcode(pos++,proc->processID,in->code);
}
//create pt
pthread=(struct process_thread*)malloc(sizeof(struct process_thread));
if(pthread==NULL) die("error alloking new thread");
pthread->IP=proc_offs+(atoi(proc->pc->org));
pthread->communication_in_a=0;
pthread->communication_out_a=0;
pthread->communication_in_b=0;
pthread->communication_out_b=0;
pthread->prev=NULL;
pthread->next=NULL;
pthread->ptask=NULL;
ptask=(struct process_task*)malloc(sizeof(struct process_task));
if(ptask==NULL) die("error alloking new task");
ptask->ID=proc->processID;
ptask->n_threads=1;
ptask->prev=NULL;
ptask->next=NULL;
ptask->primo_thread=NULL;
ptask->ultimo_thread=NULL;
ptask->cur_thread=pthread;
ptask->communication_in_a=0;
ptask->communication_out_a=0;
ptask->communication_in_b=0;
ptask->communication_out_b=0;
get_symbols(&ptask->out_symbol,&ptask->out_color);
//add pt
add_thread(pthread,ptask);
add_task(ptask);
//recalc next_offs
next_offs+=proc_offs+proc->pc->len;
//free proc&pc
in=proc->pc->first;
do{
if(in->left) free_expr(in->left);
if(in->right) free_expr(in->right);
if(in->code) free(in->code);
if(in->next) {in=in->next;free(in->prev);}
else {free(in);in=NULL;}
}while(in!=NULL);
}
xx=sqrt(size_arena);
lato_y=rint(xx);
lato_x=ceil(xx);
max_x=(int)lato_x;
max_y=(int)lato_y;
if(output_mode>=OUTPUT_DEBUG)
{
sprintf(out_str,"max_x=%d max_y=%d\n",max_x,max_y);
fputs(out_str,fpout);
}
```

```
//init_graph
if(vo_mode==VO_FRAMEBUFFER) init_txt();
if(vo_mode==VO_X11) init_x11();
}

void deinit_game()
{
if(vo_mode==VO_FRAMEBUFFER) deinit_txt();
if(vo_mode==VO_X11) deinit_x11();
if(g_actual_CPU>=CPU_cicle)
{
sprintf(out_str,"The match is a Draw !!\n");
}
else
{
sprintf(out_str,"...and the winner is process #%d        ,at %d CPU
cicles                      \n",primo_task->ID,g_actual_CPU);
if(output_mode>=OUTPUT_DEBUG2) sprintf(out_str,"...and the winner is
process #%d (%c)                       , at %d CPU cicles
\n",primo_task->ID,primo_task->out_symbol,g_actual_CPU);
}
fputs(out_str,fpout);
}
```

A.5 x11_output.h

```
void x11_cell_refresh(int addr,struct process_thread *pt);
void init_x11(void);
void deinit_x11(void);
void x11_print_arena_snap(void);
void gtk_display_curr_instr(char *ss);
void gtk_update_warrior(struct process_task *ptask);
```

A.6 x11_output.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<ctype.h>
#include<gtk/gtk.h>
#include"main.h"
#include"list_utils.h"
```

```
//#include"parse2.h"
#include"pack.h"
//#include"init_game.h"
#include"scheduler.h"
#include"execute.h"
#include"debug_output.h"
#include"txt_output.h"
#include"x11_output.h"

GtkWidget *window1,*window2,*window3;
GtkWidget *vbox;
GtkWidget *drawing_area;
GdkDrawable *main_map;
GtkWidget *hbox;
GtkWidget *liststore_instr;
GtkWidget *liststore_warrior;
GtkWidget *scrolledwindow1,*scrolledwindow2,*scrolledwindow3;
GtkWidget *statusbar;
GtkWidget *warriorview,*instrview;
GtkTreeIter iter;
gint context_id,timeout_tag,gtkhistory,gtkwarrior;
GtkListStore *model,*model2;             /* l'oggetto model   */

  GtkWidget *view,*view2;                          /* -\                */
  GtkCellRenderer *renderer;                           /* --> l'oggetto view  */
  GtkTreeSelection *selection,*selection2;      /* -/                */
struct process_task *gtktask;
static GdkPixmap *pixmap = NULL;
GdkGC* execute_gc;
GdkColor black;
GdkColormap *execute_colormap;
GdkPixbuf *pixbuf;
GdkPixmap *bullet;

struct process_task* get_cell_owner(int addr)
{
struct unpacked_op_mem mem;
struct process_task *mtask;
unpack(addr,&mem);
if(mem.processID==0) return NULL;
mtask=primo_task;
do{
if(mem.processID==mtask->ID) return mtask;
mtask=mtask->next;
}while(mtask);
```

```
//mtask=first_killed_task;
//do{
// if(mem.processID==mtask->ID) return mtask;
// mtask=mtask->next;
//}while(mtask);
return NULL;
}
static void color_icon (GdkPixbuf *pixbuf, int x, int y, guchar red, guchar
green, guchar blue, guchar alpha)
{
  int width, height, rowstride, n_channels,i,j;
  guchar *pixels, *p;

  n_channels = gdk_pixbuf_get_n_channels (pixbuf);

  g_assert (gdk_pixbuf_get_colorspace (pixbuf) == GDK_COLORSPACE_RGB);
  g_assert (gdk_pixbuf_get_bits_per_sample (pixbuf) == 8);
  g_assert (gdk_pixbuf_get_has_alpha (pixbuf));
  g_assert (n_channels == 4);

  width = gdk_pixbuf_get_width (pixbuf);
  height = gdk_pixbuf_get_height (pixbuf);

  g_assert (x >= 0 && x < width);
  g_assert (y >= 0 && y < height);

  rowstride = gdk_pixbuf_get_rowstride (pixbuf);
  pixels = gdk_pixbuf_get_pixels (pixbuf);

  for(i=0;i<10;i++)
  for(j=0;j<10;j++)
  {
  p = pixels + j * rowstride + i * n_channels;
  p[0] = red;
  p[1] = green;
  p[2] = blue;
  p[3] = alpha;
  }

  p = pixels + y * rowstride + x * n_channels;
  p[0] = red;
  p[1] = green;
  p[2] = blue;
  p[3] = alpha;
  gdk_pixbuf_fill(pixbuf,(guint32)p);
```

```
}
static guint32 get_RGBA(guchar red, guchar green, guchar blue)
{
guchar *p,*pixels;
int x=0,y=0,n_channels,rowstride;
n_channels = gdk_pixbuf_get_n_channels (pixbuf);
rowstride = gdk_pixbuf_get_rowstride (pixbuf);
   pixels = gdk_pixbuf_get_pixels (pixbuf);
   p = pixels + y * rowstride + x * n_channels;
   p[0] = red;
   p[1] = green;
   p[2] = blue;
   p[3] = 1;
return *p;
}
static void on_destroy (GtkWidget * widget, gpointer data)
{
    gtk_main_quit ();
    sprintf(out_str,"User Termination.\n");
    fputs(out_str,fpout);
    exit(1);
}
static void on_history_destroy (GtkWidget * widget, gpointer data)
{
    gtkhistory=0;
}
static void on_warrior_destroy (GtkWidget * widget, gpointer data)
{
    gtkwarrior=0;
}
void gtk_display_curr_instr(char *ss)
{
if(gtkhistory)
{
gtk_list_store_append(GTK_LIST_STORE(model), &iter);
   gtk_list_store_set(GTK_LIST_STORE(model), &iter,0, ss,-1);
   //and scoll down !!!!
   //gtk_signal_emit_by_name(GTK_OBJECT(scrolledwindow2),"scroll_event",
   NULL);
   }
}
gint gtk_statistic(gpointer data)
{

return TRUE;
```

```
}
void gtk_update_warrior(struct process_task *ptask)
{
int x,n;
char s1[MAXSTR];
n=ptask->ID - 1;
x=gtk_tree_model_iter_nth_child(GTK_TREE_MODEL(model2),&iter,NULL,n);
if(x)
{
//gtk_list_store_set(GTK_LIST_STORE(model2), &iter,0,
pixbuf,1,"ciao",2,out_str,-1);
//gtk_list_store_set(GTK_LIST_STORE(model2), &iter,0,
NULL,1,"pippo",2,"asas",-1);
pixbuf=gdk_pixbuf_new(GDK_COLORSPACE_RGB,TRUE,8,10,10);
sprintf(out_str,"#%d (%c)
color(%d,%d,%d)",ptask->ID,ptask->out_symbol,ptask->m_color.red,ptask->
m_color.green,ptask->m_color.blue);
bullet = gdk_pixmap_new (drawing_area->window, 10,10, -1);
gdk_gc_set_foreground (execute_gc, &(ptask->m_color));
gdk_draw_rectangle (bullet, execute_gc, TRUE, 0, 0,10,10);
gdk_pixbuf_get_from_drawable(pixbuf,bullet,NULL,0,0,0,0,10,10);
if(ptask->n_threads)
{
sprintf(s1,"%d",ptask->n_threads);
}
else
{
sprintf(s1,"Dead!");
}
gtk_list_store_set(GTK_LIST_STORE(model2), &iter,0,
pixbuf,1,s1,2,out_str,-1);
g_object_unref(bullet);
g_object_unref(pixbuf);
}
}
static gint main_expose (GtkWidget *widget, GdkEventExpose *event)
{
  int x1,y1,x2,y2;
  int i;
struct process_task *mtask;
  gdk_gc_set_foreground (execute_gc, &black);
  gdk_draw_rectangle (widget->window, widget->style->black_gc, TRUE,
  event->area.x, event->area.y,event->area.width,event->area.height);
for(i=0;i<size_arena;i++)
{
```

```
mtask=get_cell_owner(i);
if(mtask)x11_cell_refresh(i,mtask->cur_thread);
else x11_cell_refresh(i,NULL);
}
//gdk_draw_rectangle(widget->window,execute_gc,TRUE,10,10,5,5);
  return TRUE; /* Why do we need this??? */
}
static gint main_configure (GtkWidget *widget, GdkEventConfigure *event)
{
  main_map = widget->window;
  execute_gc = gdk_gc_new (drawing_area->window);
  execute_colormap = gdk_window_get_colormap (drawing_area->window);
  black.red = 15000;
  black.green = 15000;
  black.blue = 15000;
  gdk_color_alloc (execute_colormap, &black);
  gdk_gc_set_background (execute_gc, &black);

  return FALSE;
}
void x11_cell_refresh(int addr,struct process_thread *pt)
{
int x,y;
x=(addr%gtk_sc_x)*6;
y=(int)(((double)addr/(double)gtk_sc_x))*6;
if(pt==NULL)
{
gdk_gc_set_foreground(execute_gc,&black);
}
else
{
gdk_gc_set_foreground(execute_gc,&(pt->ptask->m_color));
}
gdk_draw_rectangle(main_map,execute_gc,TRUE,x,y,5,5);
}
gint gtk_execute(gpointer data)
{
struct process_task *task_to_kill;
int alive;
static int actual_CPU=0,infinite_CPU=0;
if(CPU_cicle==-1) {CPU_cicle=1;infinite_CPU=1;}
if(actual_CPU++>=CPU_cicle)
{
if(gtkhistory)
{
```

```
//gtk_display_draw
gtk_list_store_append(GTK_LIST_STORE(model), &iter);
sprintf(out_str,"This is a draw !!!");
   gtk_list_store_set(GTK_LIST_STORE(model), &iter,0, out_str,-1);
   }
context_id = gtk_statusbar_get_context_id(GTK_STATUSBAR
(statusbar),"my_statusbar");
gtk_statusbar_push (GTK_STATUSBAR (statusbar), context_id, "(Not
Running), this is a draw!");
return FALSE;
}
if(primo_task==ultimo_task)
{
if(gtkhistory)
{
//gtk_display_winner
gtk_list_store_append(GTK_LIST_STORE(model), &iter);
sprintf(out_str," ...and the winner is #%d (%c) after %d CPU
cicles",primo_task->ID,primo_task->out_symbol,actual_CPU);
   gtk_list_store_set(GTK_LIST_STORE(model), &iter,0, out_str,-1);
   }
context_id = gtk_statusbar_get_context_id(GTK_STATUSBAR
(statusbar),"my_statusbar");
sprintf(out_str,"(Not Running), ...and the winner is #%d (%c) after %d
CPU cicles",primo_task->ID,primo_task->out_symbol,actual_CPU);
gtk_statusbar_push (GTK_STATUSBAR (statusbar), context_id,out_str);
return FALSE;
}
alive=step(gtktask->cur_thread);
if(infinite_CPU) actual_CPU--;
if(alive)
{
gtktask->cur_thread=gtktask->cur_thread->next;
if(gtktask->cur_thread==NULL) gtktask->cur_thread=gtktask->primo_thread;
}
else
{
del_thread(gtktask->cur_thread);
if(gtkhistory)
{
//gtk_display thread dead
gtk_list_store_append(GTK_LIST_STORE(model), &iter);
sprintf(out_str,"a thread of task #%d is dead",gtktask->ID);
   gtk_list_store_set(GTK_LIST_STORE(model), &iter,0, out_str,-1);
   }
```

```
}
task_to_kill=gtktask;
gtktask=gtktask->next; if(gtktask==NULL) gtktask=primo_task;
if(task_to_kill->cur_thread==NULL)
{
if(gtkhistory)
{
gtk_list_store_append(GTK_LIST_STORE(model), &iter);
sprintf(out_str,"the task #%d is dead",task_to_kill->ID);
   gtk_list_store_set(GTK_LIST_STORE(model), &iter,0, out_str,-1);
   }
   if(gtkwarrior)
   {

   }
   gtk_update_warrior(task_to_kill);
del_task(task_to_kill);
}
return TRUE;
}
void multi_win_init()
{
window1 = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_container_set_border_width (GTK_CONTAINER (window1), 1);
        gtk_window_set_title (GTK_WINDOW (window1), "WiCE");
        //gtk_window_set_policy(GTK_WINDOW(window1),FALSE,FALSE,FALSE);
        gtk_window_set_default_size (GTK_WINDOW (window1), max_x*6, max_y*6);
        //gtk_window_set_default_icon_from_file (PIXMAPS_DIR
        "/hello-icon.gif",NULL);
        g_signal_connect (G_OBJECT (window1), "destroy",G_CALLBACK (on_destroy),
        NULL);
vbox=gtk_vbox_new(FALSE,0);
gtk_container_add(GTK_CONTAINER(window1),vbox);
gtk_widget_show(vbox);
//add draw area
drawing_area = gtk_drawing_area_new ();
     gtk_widget_set_size_request (drawing_area, max_x*6, max_y*6);
     gtk_signal_connect (GTK_OBJECT (drawing_area),
     "expose_event",(GtkSignalFunc) main_expose, NULL);
     gtk_signal_connect
     (GTK_OBJECT(drawing_area),"configure_event",(GtkSignalFunc)
     main_configure, NULL);
     gtk_widget_set_events (drawing_area, GDK_EXPOSURE_MASK);
     gtk_box_pack_start(GTK_BOX(vbox),drawing_area,TRUE,TRUE,0);
     gtk_widget_show(drawing_area);
```

```
statusbar=gtk_statusbar_new();
gtk_box_pack_start(GTK_BOX(vbox),statusbar,FALSE,FALSE,0);
gtk_widget_show(statusbar);
context_id = gtk_statusbar_get_context_id(GTK_STATUSBAR
(statusbar),"my_statusbar");
gtk_statusbar_push (GTK_STATUSBAR (statusbar), context_id, "Not Running.");

gtk_widget_show(window1);

window2 = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_container_set_border_width (GTK_CONTAINER (window2), 1);
        gtk_window_set_title (GTK_WINDOW (window2), "History");
        gtk_window_set_default_size (GTK_WINDOW (window2), 600, 200);
        g_signal_connect (G_OBJECT (window2), "destroy",G_CALLBACK
        (on_history_destroy), NULL);
        scrolledwindow2 = gtk_scrolled_window_new (NULL, NULL);
   gtk_container_add(GTK_CONTAINER(window2),scrolledwindow2);
   gtk_container_set_border_width (GTK_CONTAINER (scrolledwindow2), 10);
   gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW
   (scrolledwindow2),GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
   gtk_widget_show (scrolledwindow2);
   model     = gtk_list_store_new(1, G_TYPE_STRING);
   view      = gtk_tree_view_new_with_model (GTK_TREE_MODEL(model));
   selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(view));
     renderer = gtk_cell_renderer_text_new();
   gtk_tree_view_insert_column_with_attributes(GTK_TREE_VIEW(view),-1,"History"
   ,renderer,"text",0,NULL);
   gtk_widget_show (view);
   g_object_unref(model);
   gtk_container_add(GTK_CONTAINER(scrolledwindow2),view);
   gtk_window_move(GTK_WINDOW(window2),70,650);
        gtk_widget_show(window2);

window3 = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_container_set_border_width (GTK_CONTAINER (window3), 1);
        gtk_window_set_title (GTK_WINDOW (window3), "Warriors");
        gtk_window_set_default_size (GTK_WINDOW (window3), 200, 400);
        g_signal_connect (G_OBJECT (window3), "destroy",G_CALLBACK
        (on_warrior_destroy), NULL);
        scrolledwindow3 = gtk_scrolled_window_new (NULL, NULL);
   gtk_container_add(GTK_CONTAINER(window3),scrolledwindow3);
   gtk_container_set_border_width (GTK_CONTAINER (scrolledwindow3), 10);
   gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW
   (scrolledwindow3),GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
   gtk_widget_show (scrolledwindow3);
   model2     = gtk_list_store_new(3,
   GDK_TYPE_PIXBUF,G_TYPE_STRING,G_TYPE_STRING);
   view2      = gtk_tree_view_new_with_model (GTK_TREE_MODEL(model2));
   selection2 = gtk_tree_view_get_selection(GTK_TREE_VIEW(view2));
renderer = gtk_cell_renderer_pixbuf_new();
   gtk_tree_view_insert_column_with_attributes(GTK_TREE_VIEW(view2),-1,"<>",
   renderer,"pixbuf",0,NULL);
   renderer = gtk_cell_renderer_text_new();
   gtk_tree_view_insert_column_with_attributes(GTK_TREE_VIEW(view2),-1,"Threads
   ",renderer,"text",1,NULL);
   renderer = gtk_cell_renderer_text_new();
   gtk_tree_view_insert_column_with_attributes(GTK_TREE_VIEW(view2),-1,"
   Warriors",renderer,"text",2,NULL);
   gtk_widget_show (view2);
   g_object_unref(model2);
   gtk_container_add(GTK_CONTAINER(scrolledwindow3),view2);
   gtk_window_move(GTK_WINDOW(window3),660,150);
        gtk_widget_show(window3);
}
void one_win_init()
{
window1 = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_container_set_border_width (GTK_CONTAINER (window1), 1);
        gtk_window_set_title (GTK_WINDOW (window1), "WiCE");
        gtk_window_set_default_size (GTK_WINDOW (window1), max_x*6, max_y*6);
        //gtk_window_set_default_icon_from_file (PIXMAPS_DIR
        "/hello-icon.gif",NULL);
        g_signal_connect (G_OBJECT (window1), "destroy",G_CALLBACK (on_destroy),
        NULL);
vbox=gtk_vbox_new(FALSE,0);
gtk_container_add(GTK_CONTAINER(window1),vbox);
gtk_widget_show(vbox);
hbox=gtk_hbox_new(FALSE,0);
gtk_box_pack_start(GTK_BOX(vbox),hbox,TRUE,TRUE,0);
gtk_widget_show(hbox);
scrolledwindow1 = gtk_scrolled_window_new (NULL, NULL);
   gtk_box_pack_start(GTK_BOX(hbox),scrolledwindow1,TRUE,TRUE,0);
   gtk_container_set_border_width (GTK_CONTAINER (scrolledwindow1), 10);
   gtk_widget_set_size_request(scrolledwindow1, max_x*7, max_y*7);
   gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW
   (scrolledwindow1),GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
   gtk_widget_show (scrolledwindow1);
//area disegno
drawing_area = gtk_drawing_area_new ();
     gtk_widget_set_size_request (drawing_area, max_x*6, max_y*6);
     gtk_signal_connect (GTK_OBJECT (drawing_area),
     "expose_event",(GtkSignalFunc) main_expose, NULL);
     gtk_signal_connect
     (GTK_OBJECT(drawing_area),"configure_event",(GtkSignalFunc)
     main_configure, NULL);
     gtk_widget_set_events (drawing_area, GDK_EXPOSURE_MASK);
     gtk_scrolled_window_add_with_viewport(GTK_SCROLLED_WINDOW(
     scrolledwindow1),drawing_area);
     gtk_widget_show(drawing_area);
scrolledwindow3 = gtk_scrolled_window_new (NULL, NULL);
   gtk_box_pack_start(GTK_BOX(hbox),scrolledwindow3,FALSE,FALSE,0);
   gtk_container_set_border_width (GTK_CONTAINER (scrolledwindow3), 10);
   gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW
   (scrolledwindow3),GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
   gtk_widget_show (scrolledwindow3);
   model2     = gtk_list_store_new(3,
   GDK_TYPE_PIXBUF,G_TYPE_STRING,G_TYPE_STRING);
   view2      = gtk_tree_view_new_with_model (GTK_TREE_MODEL(model2));
   selection2 = gtk_tree_view_get_selection(GTK_TREE_VIEW(view2));
renderer = gtk_cell_renderer_pixbuf_new();
   gtk_tree_view_insert_column_with_attributes(GTK_TREE_VIEW(view2),-1,"<>",
   renderer,"pixbuf",0,NULL);
   renderer = gtk_cell_renderer_text_new();
   gtk_tree_view_insert_column_with_attributes(GTK_TREE_VIEW(view2),-1,"Threads
   ",renderer,"text",1,NULL);
   renderer = gtk_cell_renderer_text_new();
   gtk_tree_view_insert_column_with_attributes(GTK_TREE_VIEW(view2),-1,"
   Warriors",renderer,"text",1,NULL);
   gtk_widget_show (view2);
   g_object_unref(model2);
   gtk_widget_set_size_request(view2, 150, 200);
   gtk_container_add(GTK_CONTAINER(scrolledwindow3),view2);
   //gtk_window_move(GTK_WINDOW(window3),960,300);
        gtk_widget_show(window3);
scrolledwindow2 = gtk_scrolled_window_new (NULL, NULL);
   gtk_box_pack_start(GTK_BOX(vbox),scrolledwindow2,FALSE,FALSE,0);
   gtk_container_set_border_width (GTK_CONTAINER (scrolledwindow2), 10);
   gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW
   (scrolledwindow2),GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
   gtk_widget_show (scrolledwindow2);
   model     = gtk_list_store_new(1, G_TYPE_STRING);
   view      = gtk_tree_view_new_with_model (GTK_TREE_MODEL(model));
   selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(view));
     renderer = gtk_cell_renderer_text_new();
   gtk_tree_view_insert_column_with_attributes(GTK_TREE_VIEW(view),-1,"History"
   ,renderer,"text",0,NULL);
   gtk_widget_show (view);
   g_object_unref(model);
   gtk_widget_set_size_request(view, 100, 200);
   gtk_container_add(GTK_CONTAINER(scrolledwindow2),view);
   //gtk_window_move(GTK_WINDOW(window2),90,700);
        gtk_widget_show(window2);
//gtk_signal_connect (GTK_OBJECT (view), "scroll_event",(GtkSignalFunc)
my_get_scroll, NULL);
statusbar=gtk_statusbar_new();
gtk_box_pack_start(GTK_BOX(vbox),statusbar,FALSE,FALSE,0);
gtk_widget_show(statusbar);
context_id = gtk_statusbar_get_context_id(GTK_STATUSBAR
(statusbar),"my_statusbar");
gtk_statusbar_push (GTK_STATUSBAR (statusbar), context_id, "Not Running.");
        gtk_widget_show(window1);
}
void init_x11()
{
GtkWidget *label;
struct process_task *ptask;
guint32 m_RGBA;
gtk_init(NULL,NULL);
gtkhistory=1;
gtkwarrior=1;
gtk_sc_x=max_x;
gtk_sc_y=max_y;
if(b_multi_win)
{
multi_win_init();
}
else
{
one_win_init();
}
for(ptask=primo_task;ptask;ptask=ptask->next)
{
do {
ptask->m_color.red = random() % 65535;
ptask->m_color.green = random() % 65535;
ptask->m_color.blue = random() % 65535;
} while
(ptask->m_color.red+ptask->m_color.green+ptask->m_color.blue<100000);
gdk_color_alloc (execute_colormap, &(ptask->m_color));
}
```

```
ptask=primo_task;
if(gtkwarrior)
{
do {
gtk_list_store_append(GTK_LIST_STORE(model2), &iter);
pixbuf=gdk_pixbuf_new(GDK_COLORSPACE_RGB,TRUE,8,10,10);
//color_icon(pixbuf,0,0,ptask->m_color.red,ptask->m_color.green,
ptask->m_color.blue,1);
//m_RGBA=get_RGBA(ptask->m_color.red,ptask->m_color.green,ptask->
m_color.blue);
//gdk_pixbuf_fill(pixbuf,m_RGBA);
sprintf(out_str,"#%d (%c)
color(%d,%d,%d)",ptask->ID,ptask->out_symbol,ptask->m_color.red,
ptask->m_color.green,ptask->m_color.blue);
bullet = gdk_pixmap_new (drawing_area->window, 10,10, -1);
   gdk_gc_set_foreground (execute_gc, &(ptask->m_color));
   gdk_draw_rectangle (bullet, execute_gc, TRUE, 0, 0,10,10);
   gdk_pixbuf_get_from_drawable(pixbuf,bullet,NULL,0,0,0,0,10,10);
   gtk_list_store_set(GTK_LIST_STORE(model2), &iter,0,
   pixbuf,1,"1",2,out_str,-1);
   g_object_unref(bullet);
   g_object_unref(pixbuf);
   ptask=ptask->next;
   }while(ptask);
   }
   // set gtk_timeout_add
   timeout_tag=gtk_timeout_add(gtkwaittime,gtk_execute,NULL);
   //timeout_tag=gtk_timeout_add(STAT_WAIT_TIME,gtk_statistic,NULL);
   context_id = gtk_statusbar_get_context_id(GTK_STATUSBAR
   (statusbar),"my_statusbar");
gtk_statusbar_push (GTK_STATUSBAR (statusbar), context_id, "Running....");
gtktask=get_first();
        gtk_main ();
}
void deinit_x11()
{
//gtk_exit(0);
}
void x11_print_arena_snap(void)
{

}
```

A.7 txt_output.h

```
void curses_cell_refresh(int addr,struct process_thread *pt);
void init_txt(void);
void deinit_txt(void);
void curses_print_arena_snap(void);
```

A.8 txt_output.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<ctype.h>
#include<ncurses.h>
#include<math.h>
#include<gtk/gtk.h>
#include"main.h"
#include"list_utils.h"
//#include"parse2.h"
#include"pack.h"
//#include"init_game.h"
//#include"scheduler.h"
#include"execute.h"
#include"debug_output.h"
#include"txt_output.h"
#include"x11_output.h"

void curses_cell_refresh(int addr,struct process_thread *pt)
{
int x,y,col;
char car;
addr2coords(addr,&x,&y);
get_p_attr(pt->ptask->ID,&car,&col);
mvaddch(y+yd,x+xd,car);
refresh();
}
void init_txt()
{
int count,curs_x,curs_y,col;
char car;
struct unpacked_op_mem mem;
struct process_task *ptask;
xd=1; /*x offset from beginnig of screen*/
```

```
yd=1; /*y offest from beginning of screen*/
xl=25; /*x space from end of screen*/
yl=1; /*y space from end of screen*/
initscr();
cbreak();
noecho();
sc_x=COLS-xd-xl;
max_sc_y=LINES-yd-yl;
sc_y=(int)ceil((double)size_arena/(double)sc_x);
txt_y_warrior_list=sc_x+4;
if(output_mode>=OUTPUT_DEBUG)
{
sprintf(out_str,"sc_x=%d sc_y=%d COLS=%d ROWS=%d(max
%d)\n",sc_x,sc_y,COLS,LINES,max_sc_y);
fputs(out_str,fpout);
}
if(sc_y>(LINES-yd-yl))
{
deinit_txt();
sprintf(out_str,"arena y (%d) excedes max video lines (%d)...switching
to VO_NONE mode\n",sc_y,LINES);
fputs(out_str,fpout);
vo_mode=VO_NONE;
return;
}
curs_y=0;
for(count=0;count<size_arena;count++)
{
unpack(count,&mem);
get_p_attr(mem.processID,&car,&col);
addr2coords(count,&curs_x,&curs_y);
mvaddch(curs_y+yd,curs_x+xd,car);
}
// make box
mvaddch(0,0,'+');
for(count=0;count<sc_x;count++) addch('=');
addch('+');
mvaddch(sc_y+1,0,'+');
for(count=0;count<sc_x;count++) addch('=');
addch('+');
for(count=0;count<sc_y;count++)
{
mvaddch(count+yd,0,'|');
mvaddch(count+yd,sc_x+1,'|');
}
```

```
//print warriors list
for(ptask=primo_task;ptask=ptask;ptask=ptask->next)
{

sprintf(out_str,"V #%d (%c)",ptask->ID,ptask->out_symbol);
mvaddstr(ptask->ID,txt_y_warrior_list,out_str);
}
//
refresh();
}
void deinit_txt()
{
move(sc_y+3+yd,0);
printw("leaving...");
if(g_actual_CPU>=CPU_cicle)
{
sprintf(out_str,"The match is a Draw!!");
}
else
{
sprintf(out_str,"...and the winner is process #%d (%c)  , at %d CPU
cicles
\n",primo_task->ID,primo_task->out_symbol,g_actual_CPU);
}
printw(out_str);
getch();
endwin();
}
void curses_print_arena_snap(void)
{

}
```

A.9 scheduler.h

```
void play_game(void);
struct process_task* get_first(void);
```

A.10 scheduler.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
```

```
#include<errno.h>
#include<ctype.h>
#include<gtk/gtk.h>
#include"main.h"
#include"list_utils.h"
#include"parse2.h"
#include"pack.h"
#include"init_game.h"
#include"scheduler.h"


struct process_task* get_first()
{
return primo_task;
}
int step(struct process_thread *pt)
{
struct unpacked_op_mem *code;
int alive,old_ip;
code=(struct unpacked_op_mem*)malloc(sizeof(struct unpacked_op_mem));
if(code==NULL) die("error alloking getting opcode from mem_array");
unpack(pt->IP,code);
old_ip=pt->IP;
alive=execute(code,pt);
if(alive)
{
code->processID=pt->ptask->ID;
pack2mem(old_ip,code);
}
return alive;
}
void play_game()
{
struct process_task *ptask,*task_to_kill;
int actual_CPU,alive,infinite_CPU=0;
ptask=get_first();
//init_graph()
actual_CPU=0;
if(CPU_cicle==-1) {CPU_cicle=1;infinite_CPU=1;}
while(actual_CPU++<CPU_cicle)
{
if(infinite_CPU) actual_CPU--;
//control if there is a winner
if(primo_task==ultimo_task) break;
if(sleeptime) sleep(sleeptime);
alive=step(ptask->cur_thread);
if(alive)
{
ptask->cur_thread=ptask->cur_thread->next;
if(ptask->cur_thread==NULL) ptask->cur_thread=ptask->primo_thread;
}
else
{
del_thread(ptask->cur_thread);
if(output_mode>=OUTPUT_DEBUG)
{
sprintf(out_str,"proc #%d: thread killed.                              \n",ptask->ID);
if(vo_mode==VO_NONE && log_mode) fputs(out_str,fpout);
if(vo_mode==VO_FRAMEBUFFER) mvaddstr(sc_y+2+yd,0,out_str);
}
}
task_to_kill=ptask;
ptask=ptask->next; if(ptask==NULL) ptask=primo_task;
if(task_to_kill->cur_thread==NULL)
{
if(output_mode>=OUTPUT_DEBUG)
{
sprintf(out_str,"process killed.                                                \n");
if(vo_mode==VO_NONE && log_mode) fputs(out_str,fpout);
if(vo_mode==VO_FRAMEBUFFER) mvaddstr(sc_y+2+yd,0,out_str);
}
if(vo_mode==VO_FRAMEBUFFER)
{
mvaddch(task_to_kill->ID,txt_y_warrior_list,'X');
sprintf(out_str,"process killed.                                                \n");
mvaddstr(sc_y+2+yd,0,out_str);
}
del_task(task_to_kill);
}
}
g_actual_CPU=actual_CPU;
}
```

A.11 parse2.h

```
unsigned int solve_vt(char *name,struct Process *proc);
unsigned int solve_expr(struct expr_node *expr,struct Process *proc);
void generate_code(struct Process *proc);
```

A.12 parse2.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<ctype.h>
#include<gtk/gtk.h>
#include"main.h"
#include"list_utils.h"
#include"parse2.h"
#include"pack.h"

unsigned int solve_vt(char *name,struct Process *proc)
{
unsigned int val;
struct var_table *vt;
for(vt=proc->pc->vt_first;vt;vt=vt->next)
{
if(((strcmp(name,vt->name))==0) || ((strcmp(name,vt->name+1))==0))
{
val=solve_expr(vt->val_first,proc);
if(vt->name[0]==':'){val=val-current_make_node;}  // adjust for labels
return val;
}
}
return -1;
}
unsigned int solve_expr(struct expr_node *expr,struct Process *proc)
{
int val,lval,rval;
if(expr->type==NUMBER)
{
val=atoi(expr->str);
return (val%size_arena);
}
if(expr->type==VAR)
{
val=solve_vt(expr->str,proc);
return (val%size_arena);
}
if(expr->type){

if(expr->left)
```

```
{
lval=solve_expr(expr->left,proc);
}
if(expr->right)
{
rval=solve_expr(expr->right,proc);
}
if((strcmp("+",expr->str))==0)
{
val=(lval+rval)%size_arena;
return val;
}
if((strcmp("-",expr->str))==0)
{
val=(lval-rval)%size_arena;
return val;
}
if((strcmp("*",expr->str))==0)
{
val=(lval*rval)%size_arena;
return val;
}
if((strcmp("/",expr->str))==0)
{
val=(lval/rval)%size_arena;
return val;
}
if((strcmp("%",expr->str))==0)
{
val=(lval%rval)%size_arena;
return val;
}
if((strcmp("==",expr->str))==0)
{
val=lval==rval;
return val;
}
if((strcmp("!=",expr->str))==0)
{
val=lval!=rval;
return val;
}
if((strcmp("<",expr->str))==0)
{
val=lval<rval;
return val;
}
if((strcmp(">",expr->str))==0)
{
val=lval>rval;
return val;
}
if((strcmp("<=",expr->str))==0)
{
val=lval<=rval;
return val;
}
if((strcmp(">=",expr->str))==0)
{
val=lval>=rval;
return val;
}

}
}
void compute_asserts(struct Process *proc)
{
struct var_table *vt;
struct expr_node *expr;
unsigned int ret_val;
for(vt=proc->pc->vt_first;vt;vt=vt->next)
{
if((strcmp(vt->name,ASSERT_STR))==0)
{
ret_val=solve_expr(vt->val_first,proc);
if(!ret_val)
{
die("assert condition failed");
}
}
}
}
void generate_code(struct Process *proc)
{
struct instruction_node *in;
struct unpacked_op_mem op_mem;
void *new_code;
char s1[MAXSTR];
int x;
compute_asserts(proc);
```

```
for(in=proc->pc->first;in;in=in->next)
{
op_mem.processID=proc->processID;
op_mem.opcode=str_to_code(in->instr);
op_mem.a_pref=str_to_code(in->laddr);
op_mem.b_pref=str_to_code(in->raddr);
if((strcmp(in->modifier,"NULL"))==0) {get_default_mod(&op_mem);} else
{op_mem.mod=str_to_code(in->modifier);}
current_make_node=in->num_node;
if(in->left) {op_mem.a_val=solve_expr(in->left,proc);} else
{op_mem.a_val=0;}
if(in->right) {op_mem.b_val=solve_expr(in->right,proc);} else
{op_mem.b_val=0;}

//malloc void/struct
if(arena_mem_type==MEM_TYPE_ONE) new_code=(struct
array_mem_small*)malloc(sizeof(struct array_mem_small));
if(arena_mem_type==MEM_TYPE_TWO) new_code=(struct
array_mem_mid*)malloc(sizeof(struct array_mem_mid));
if(arena_mem_type==MEM_TYPE_FOUR) new_code=(struct
array_mem_norm*)malloc(sizeof(struct array_mem_norm));
if(new_code==NULL)
{sprintf(s1,"at line %d ",in->line_count);die("error malloking new_code struct");}
//jump adjust
//if((op_mem.opcode==op_JMP)||(op_mem.opcode==op_JMZ)||(op_mem.opcode==
op_JMN)||(op_mem.opcode==op_DJN))
//{
// op_mem.a_val=(op_mem.a_val-in->num_node)%size_arena;
//}
//pack
pack(&op_mem,new_code);
//add code_node
in->code=new_code;
}
//solve_org
if((strcmp(proc->pc->org,""))!=0)
{
current_make_node=0;
x=solve_vt(proc->pc->org,proc);
sprintf(proc->pc->org,"%d",x);
}
}
```

A.13 parse.h

```
void read_file(char *filename,struct Process **pproc);
void parse(char *filename);
void visit_tree(struct expr_node *expr);
void print_data(struct Process *proc);
```

A.14 parse.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<ctype.h>
#include<gtk/gtk.h>
#include"main.h"
#include"list_utils.h"


char my_token[MAXSTR];

char* skip_space(char *p)
{
while ((*p==' ')||(*p=='\t')) p++;
return p;
}
char* get_word(char *p)
{
int n=0;
while(isalnum(*p))
{
my_token[n++]=*p;
p++;
}
my_token[n]='\0';
return p;
}
char* get_num(char *p)
{
int n=0;
if(*p=='-') {my_token[n++]=*p;p++;p=skip_space(p);}
while(isdigit(*p))
{
my_token[n++]=*p;
```

```
p++;
}
my_token[n]='\0';
return p;
}
char* get_sym(char *p)
{
int n=0;
my_token[n]=*p;
p++;n++;
if(my_token[0]=='=')
{
if(*p=='=') {my_token[n]=*p;p++;n++;}
}
if(my_token[0]=='!')
{
if(*p=='=') {my_token[n]=*p;p++;n++;}
}
my_token[n]='\0';
return p;
}
char* get_token(char *p)
{
if(isalpha(*p)) {p=get_word(p);} else
if((isdigit(*p))||((*p=='-')&&(isdigit(*(p+1))))) {p=get_num(p);}
else {p=get_sym(p);}
return p;
}

void take_assert(char *p)
{}
int take_instr1(char *token,char *p)
{}
void take_instr2(char *token,char *p)
{}
void take_equ(char *token,char *p)
{}
int is_instr(char *token,int line_count)
{
static char *valid_instr[]={
"mov","2","dat","2","nop","0","add","2","sub","2","mul","2","div","2","mod",
"2","jmp","1",
"jmz","1","jmn","1","djn","1","spl","1","cmp","2","seq","2","sne","2","slt",
"2","ldp","2",
"stp","2","ctin","2","ctout","2","cpin","2","cpout","2"
```

```
,"NULL","0"
};
int n;
char s1[MAXSTR];
for(n=0;;n+=2)
{
if((strcasecmp(valid_instr[n],"NULL"))==0) break;
if((strcasecmp(valid_instr[n],token))==0)
return(atoi(valid_instr[n+1]));
}
//sprintf(s1,"invalid instruction at line %d .",line_count);
//die(s1);
return -1;
}
char* get_addr_mode(char *p)
{
char ss[]="#$*@{}<>";
int n;
my_token[0]='$';my_token[1]='\0';
for(n=0;n<=7;n++)
if(*p==ss[n]){my_token[0]=*p;my_token[1]='\0';p++;break;}
return p;
}
int get_processID()
{
static int p=0;
return (++p);
}
void is_modifier(char *token,int line_count)
{
static char *valid_modifier[]={
"a","b","ab","ba","f","x","i","NULL"
};
int n;
char s1[MAXSTR];
for(n=0;;n++)
{
if((strcasecmp(valid_modifier[n],"NULL"))==0) break;
if((strcasecmp(valid_modifier[n],token))==0) return ;
}
sprintf(s1,"invalid modifier at line %d .",line_count);
die(s1);
}
int is_op(int line_count)
{
```

```
char s1[MAXSTR];
if(my_token[0]=='+') return OP_ADD;
if(my_token[0]=='-') return OP_ADD;
if(my_token[0]=='*') return OP_MUL;
if(my_token[0]=='/') return OP_MUL;
if(my_token[0]=='%') return OP_MUL;
sprintf(s1,"parse error at line %d. not a valid operation
symbol",line_count);
die(s1);
}
int is_bool_op(char *p,int line_cout)
{
p=get_token(p);
if((strcmp(my_token,"=="))==0) return 1;
if((strcmp(my_token,"!="))==0) return 1;
if((strcmp(my_token,"<="))==0) return 1;
if((strcmp(my_token,">="))==0) return 1;
if((strcmp(my_token,">"))==0) return 1;
if((strcmp(my_token,"<"))==0) return 1;
if((strcmp(my_token,"="))==0) return 1;

return 0;
}
int is_item(int line_count)
{
char s1[MAXSTR];
if((isdigit(my_token[0]))||((my_token[0]=='-')&&(isdigit(my_token[1]))))
return NUMBER;
if(isalpha(my_token[0])) return VAR;
sprintf(s1,"parse error at line %d. not a valid number or
variable",line_count);
die(s1);
}
char* get_arg(struct expr_node **expr,char *p,int line_count)
{
int m_type;
struct expr_node *new_expr,*new_expr2,*save_expr,*save_tree;
char s1[MAXSTR];
p=get_token(p);
m_type=is_item(line_count);
new_expr=(struct expr_node*)malloc(sizeof(struct expr_node));
if(new_expr==NULL)
{ sprintf(s1,"at line %d, error malloc exr_node",line_count);die(s1);}
new_expr->type=m_type;
strncpy(new_expr->str,my_token,MAXSTR);
```

```
new_expr->left=NULL;
new_expr->right=NULL;
p=skip_space(p);
if((*p=='\n')||(*p==',')|| (is_bool_op(p,line_count)))
{
*expr=new_expr;
return p;
}
p=get_token(p);
m_type=is_op(line_count);
new_expr2=(struct expr_node*)malloc(sizeof(struct expr_node));
if(new_expr2==NULL)
{ sprintf(s1,"at line %d, error malloc exr_node",line_count);die(s1);}
new_expr2->type=m_type;
strncpy(new_expr2->str,my_token,MAXSTR);
new_expr2->left=new_expr;
new_expr2->right=NULL;
*expr=new_expr2;
p=skip_space(p);
p=get_arg(&(new_expr2->right),p,line_count);
if((new_expr2->type<new_expr2->right->type)&&(new_expr2->type<3)&&(new_expr2
->right->type<3))
{
save_expr=new_expr2->right;
save_tree=new_expr2->right->left;
new_expr2->right->left=new_expr2;
save_expr->left->right=save_tree;
*expr=save_expr;
}
return p;
}
char* get_b_arg(struct expr_node **expr,char *p,int line_count)
{
struct expr_node *new_expr,*new_expr2,*new_expr3;
char s1[MAXSTR];
p=get_arg(&new_expr,p,line_count);
p=skip_space(p);
if((*p=='\n')||(*p==','))
{
*expr=new_expr;
return p;
}
//if is_bool_op(p)
if(is_bool_op(p,line_count))
{
```

```
new_expr2=(struct expr_node*)malloc(sizeof(struct expr_node));
if(new_expr2==NULL)
{ sprintf(s1,"at line %d, error malloc exr_node",line_count);die(s1);}
new_expr2->type=OP_BOOL;
p=get_token(p);
strncpy(new_expr2->str,my_token,MAXSTR);
new_expr2->left=new_expr;
new_expr2->right=NULL;
p=skip_space(p);
p=get_arg(&new_expr3,p,line_count);
new_expr2->right=new_expr3;
*expr=new_expr2;
}
return p;
}
void insert_label(char *label2,int label_val,int line_num,struct Process *proc)
{
struct var_table *vt,*new_vt;
struct expr_node *expr;
char s1[MAXSTR],label[MAXSTR];
sprintf(label,":%s",label2);
for(vt=proc->pc->vt_first;vt;vt=vt->next)
if((strcmp(vt->name,label))==0)
{
sprintf(s1,"sorry, the label at line %d already exists",line_num);
die(s1);
}
new_vt=(struct var_table*)malloc(sizeof(struct var_table));
if(new_vt==NULL) die("error allocating new_vt");
expr=(struct expr_node*)malloc(sizeof(struct expr_node));
if(expr==NULL) die("error allcating expr");
strncpy(new_vt->name,label,MAXSTR);
new_vt->val_first=expr;
new_vt->val_last=expr;
expr->left=NULL;
expr->right=NULL;
expr->type=NUMBER;
sprintf(expr->str,"%d",label_val);
add_vt(proc,new_vt);
}
void insert_in_vt(char *varname,struct expr_node *expr,int line_num,struct
Process *proc)
{
struct var_table *vt,*new_vt;
//struct expr_node *expr;
```

```
char s1[MAXSTR];
if((strcmp(varname,ASSERT_STR))!=0)
for(vt=proc->pc->vt_first;vt;vt=vt->next)
if((strcmp(vt->name,varname))==0)
{
sprintf(s1,"sorry, the variable at line %d already exists",line_num);
die(s1);
}
new_vt=(struct var_table*)malloc(sizeof(struct var_table));
if(new_vt==NULL) die("error allocating new_vt");
//expr=(struct expr_node*)malloc(sizeof(struct expr_node));
//if(expr==NULL) die("error allcating expr");
strncpy(new_vt->name,varname,MAXSTR);
new_vt->val_first=expr;
new_vt->val_last=expr;
//expr->left=expr;
//expr->right=NULL;
//expr->type=NUMBER;
//sprintf(expr->str,"%d",label_val);
add_vt(proc,new_vt);
}
void add_environment_costant(char *env_name,int val,struct Process *proc)
{
struct var_table *vt;
struct expr_node *expr;
vt=(struct var_table*)malloc(sizeof(struct var_table));
if(vt==NULL) die("error allocating vt in add_environment_contants");
expr=(struct expr_node*)malloc(sizeof(struct expr_node));
if(expr==NULL) die("error on allocating exprin add_environment_contants");
expr->left=NULL;
expr->right=NULL;
expr->type=NUMBER;
sprintf(expr->str,"%d",val);
strncpy(vt->name,env_name,MAXSTR);
vt->val_first=expr;
vt->val_last=expr;
add_vt(proc,vt);
}
void add_environment_costants(struct Process *proc)
{
add_environment_costant("CORESIZE",size_arena,proc);
add_environment_costant("WARRIORS",warriors,proc);
add_environment_costant("MAXPROCESSES",maxprocesses,proc);
add_environment_costant("MAXCYCLES",CPU_cicle,proc);
add_environment_costant("MAXLENGTH",max_prog_size,proc);
add_environment_costant("MINDISTANCE",min_distance,proc);
add_environment_costant("VERSION",version,proc);
}
void read_file(char *filename,struct Process **pproc)
{
FILE *fp;
char *endfile,line[MAXSTR],*p,save_token[MAXSTR];
int label_bool=0,line_count=0,num_code=0,n_args;
struct instruction_node *new_instr;
struct Process *proc;
struct expr_node *new_expr;
proc=(struct Process*)malloc(sizeof(struct Process));
if(proc==NULL) die("errore nell'allocare struct Process");
proc->pt=NULL;
proc->pc=NULL;
proc->prev=NULL;
proc->next=NULL;
proc->processID=get_processID();
fp=fopen(filename,"r");
if(fp==NULL) die("error opening file");
proc->pc=(struct process_construct*)malloc(sizeof(struct
process_construct));
if(proc->pc==NULL) die("errore nell'allocare process_construct");
proc->pc->first=NULL;
proc->pc->last=NULL;
proc->pc->len=0;
proc->pc->org[0]='\0';
proc->pc->vt_first=NULL;
proc->pc->vt_last=NULL;
while((endfile=fgets(line,MAXSTR,fp))!=NULL)
{
line_count++;
p=&line[0];
if(*p==';') continue;
p=skip_space(p);
if(*p=='\n') continue;
p=get_token(p);
p=skip_space(p);
if((strcmp(my_token,"org"))==0)
{
p=get_word(p);strncpy(proc->pc->org,my_token,MAXSTR);
if(*p!='\n') {sprintf(save_token,"parse error at line %d. not an end
line after the org argument",line_count);die(save_token);}
continue;
}
```

```
if((strcmp(my_token,"end"))==0)
{
p=skip_space(p);
if(*p!='\n') {sprintf(save_token,"parse error after 'end' at line
%d.",line_count);die(save_token);}
break;
}
if((strcmp(my_token,"assert"))==0)
{
take_assert(p);
new_expr=(struct expr_node*)malloc(sizeof(struct expr_node));
if(new_expr==NULL)
{printf("at line %d, ",line_count);die("error allocating
new_expr");}
p=skip_space(p);
p=get_b_arg(&new_expr,p,line_count);
insert_in_vt(ASSERT_STR,new_expr,line_count,proc);
continue;
}
if(*p==':')
{
insert_label(my_token,num_code,line_count,proc);p=skip_space(++p);
if(*p=='\n') continue;
p=get_token(p);
}
n_args=is_instr(my_token,line_count);
if(n_args!=-1)
{
new_instr=(struct instruction_node*)malloc(sizeof(struct
instruction_node));
if(new_instr==NULL) {printf("at line %d , ",line_count);die("error
allocating new_instr");}
strncpy(new_instr->instr,my_token,MAXSTR);
new_instr->num_node=num_code++;
new_instr->line_count=line_count;
new_instr->prev=NULL;
new_instr->next=NULL;
new_instr->code=NULL;
new_instr->left=NULL;
new_instr->right=NULL;
new_instr->laddr[0]='#';new_instr->laddr[1]='\0';
new_instr->raddr[0]='#';new_instr->raddr[1]='\0';
strcpy(new_instr->modifier,"NULL");
if(*p=='.'){
p=get_word(++p);is_modifier(my_token,line_count);
strncpy(new_instr->modifier,my_token,MAXMOD);
}
p=skip_space(p);
if(n_args>0)
{
p=get_addr_mode(p); //$ by default. the result is in my_token
new_instr->laddr[0]=my_token[0];
new_instr->laddr[1]='\0';
//p=get_token(p);
//new_instr->left=(struct expr_node*)malloc(sizeof(struct
expr_node));
//if(new_instr->left==NULL){
//printf("at line %d , ",line_count);die("error alocating left
expr");}
p=get_arg(&new_instr->left,p,line_count);
p=skip_space(p);
if(n_args>1)
{
if(*p!=',')
{printf("at line %d , ",line_count);die("a comma expected
(,)");}
p=skip_space(++p);
p=get_addr_mode(p); //$ by default. the result is in
my_token
new_instr->raddr[0]=my_token[0];
new_instr->raddr[1]='\0';
//p=get_token(p);
//new_instr->right=(struct expr_node*)malloc(sizeof(struct
expr_node));
//if(new_instr->right==NULL){
//printf("at line %d , ",line_count);die("error allocating
right expr");}
p=get_arg(&new_instr->right,p,line_count);
p=skip_space(p);
}
}
if(*p!='\n')
{printf("at line %d , ",line_count);die("not an ending line after
command");}
//add the node
add_node(new_instr,proc);
continue;
}
strncpy(save_token,my_token,MAXSTR);
p=get_token(p);
```

```
if((strcmp(my_token,"equ"))==0)
{
new_expr=(struct expr_node*)malloc(sizeof(struct expr_node));
if(new_expr==NULL)
{printf("at line %d, ",line_count);die("error allocating
new_expr");}
p=skip_space(p);
p=get_arg(&new_expr,p,line_count);
insert_in_vt(save_token,new_expr,line_count,proc);
continue;
}
}
fclose(fp);
proc->pc->len=num_code;
add_proc(proc);
*pproc=proc;
}
void visit_tree(struct expr_node *expr)
{
if(expr)
{
sprintf(out_str,"%s",expr->str);fputs(out_str,fpout);
if(expr->type>2) return;
sprintf(out_str,"(");fputs(out_str,fpout);
visit_tree(expr->left);
sprintf(out_str,",");fputs(out_str,fpout);
visit_tree(expr->right);
sprintf(out_str,")");fputs(out_str,fpout);

}
}
void print_data(struct Process *proc)
{
struct process_construct *pc;
struct instruction_node *in;
struct expr_node *expr;
struct var_table *vt;
if(!DO_DEBUG) return;
for(in=proc->pc->first;in;in=in->next)
{
sprintf(out_str,"{line(%d[%d]),%s",in->num_node,in->line_count,in->instr
);fputs(out_str,fpout);
if(in->left)
{
sprintf(out_str,"(%c ",*in->laddr);fputs(out_str,fpout);
visit_tree(in->left);
if(in->right)
{
sprintf(out_str,",%c ",*in->raddr);fputs(out_str,fpout);
visit_tree(in->right);
}
sprintf(out_str,")");fputs(out_str,fpout);
}
sprintf(out_str,"};");fputs(out_str,fpout);
}
sprintf(out_str,"\n---var table---\n");fputs(out_str,fpout);
for(vt=proc->pc->vt_first;vt;vt=vt->next)
{
sprintf(out_str,"%s
%s",vt->name,vt->val_first->str);fputs(out_str,fpout);
if(vt->val_first->left)
{
sprintf(out_str,"(");fputs(out_str,fpout);
visit_tree(vt->val_first->left);
if(vt->val_first->right)
{
sprintf(out_str,",");fputs(out_str,fpout);
visit_tree(vt->val_first->right);
}
sprintf(out_str,")");fputs(out_str,fpout);
}
sprintf(out_str,"\n");fputs(out_str,fpout);
}
}
void parse(char *filename)
{
struct Process *proc;
//proc=(struct Process*)malloc(sizeof(struct Process));
//if(proc==NULL) die("errore nell'allocare struct Process");
//proc->pt=NULL;
//proc->pc=NULL;
//proc->prev=NULL;
//proc->next=NULL;
proc=NULL;
read_file(filename,&proc);
add_environment_costants(proc);
if(output_mode>=OUTPUT_DEBUG) print_data(proc);
generate_code(proc);
}
```

A.15 pack.h

```
#define op_NOP 1
#define op_DAT 0
#define op_MOV 2
#define op_ADD 3
#define op_SUB 4
#define op_MUL 5
#define op_DIV 6
#define op_MOD 7
#define op_JMP 8
#define op_JMZ 9
#define op_JMN 10
#define op_DJN 11
#define op_SPL 12
#define op_CMP 13
#define op_SEQ 14
#define op_SNE 15
#define op_SLT 16
#define op_LDP 17
#define op_STP 18
#define op_CTIN 19
#define op_CTOUT 20
#define op_CPIN 21
#define op_CPOUT 22
#define op_IMM 0
// #
#define op_DIR 2
// $
#define op_A_IND 3
// *
#define op_B_IND 4
// @
#define op_A_IND_PREDEC 5
// {
#define op_B_IND_PREDEC 6
// <
#define op_A_IND_POSTINC 7
// }
#define op_B_IND_POSTINC 1
// >
#define op_A 0
#define op_B 1
#define op_AB 2
#define op_BA 3
```

```
#define op_F 4
#define op_X 5
#define op_I 6

int str_to_code(char *str);
void get_default_mod(struct unpacked_op_mem *op);
void pack(struct unpacked_op_mem *in,void *out);
void unpack(int ip,struct unpacked_op_mem *out);
void pack2mem(int ip,struct unpacked_op_mem *in);
```

A.16 pack.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<ctype.h>
#include<gtk/gtk.h>
#include"main.h"
#include"list_utils.h"
#include"parse2.h"
#include"pack.h"

int str_to_code(char *str)
{
static char *conv_table[]={
"nop","1","dat","0","mov","2","add","3","sub","4","mul","5","div","6","mod",
"7","jmp","8",
"jmz","9","jmn","10","djn","11","spl","12","cmp","13","seq","14","sne","15",
"slt","16","ldp","17",
"stp","18","ctin","19","ctout","20","cpin","21","cpout","22"
,"#","0","$","2","*","3","@","4","{","5","<","6","}","7",">","1"
,"a","0","b","1","ab","2","ba","3","f","4","x","5","i","6"
,"NULL","-1"
};
int n,retval;
for(n=0;;n+=2)
{
if((strcasecmp(conv_table[n],"NULL"))==0) break;
if((strcasecmp(conv_table[n],str))==0)
{
retval=atoi(conv_table[n+1]);
return retval;
}
}
return -1;

}
void get_default_mod(struct unpacked_op_mem *op)
{
switch(op->opcode)
{
case op_NOP:
case op_DAT:
op->mod=op_F;
break;
case op_MOV:
case op_SEQ:
case op_SNE:
case op_CMP:
op->mod=op_I;
if(op->b_pref==op_IMM) op->mod=op_B;
if(op->a_pref==op_IMM) op->mod=op_AB;
break;
case op_ADD:
case op_SUB:
case op_MUL:
case op_DIV:
case op_MOD:
op->mod=op_F;
if(op->b_pref==op_IMM) op->mod=op_B;
if(op->a_pref==op_IMM) op->mod=op_AB;
break;
case op_SLT:
case op_LDP:
case op_STP:
op->mod=op_B;
if(op->a_pref==op_IMM) op->mod=op_AB;
break;
case op_JMP:
case op_JMZ:
case op_DJN:
case op_SPL:
op->mod=op_B;
break;
case op_CTIN:
case op_CTOUT:
case op_CPIN:
```

```
case op_CPOUT:
op->mod=op_F;
break;
default:
break;
}
}
void pack(struct unpacked_op_mem *in,void *out)
{
struct array_mem_small *msmall;
struct array_mem_mid *mmid;
struct array_mem_norm *mlarge;

if(arena_mem_type==MEM_TYPE_ONE)
{
msmall=(struct arena_mem_small*)out;
msmall->mem=(in->processID<<28)+(in->opcode<<23)+(in->mod<<19)+(in->
a_pref<<16)+(in->b_pref<<13)+(in->a_val<<7)+(in->b_val<<1);
}
if(arena_mem_type==MEM_TYPE_TWO)
{
mmid=(struct arena_mem_mid*)out;
mmid->processID_opcode=(in->processID<<16)+(in->opcode<<12)+(in->mod<<8)
+(in->a_pref<<5)+(in->b_pref<<2);
mmid->arg1_arg2=(in->a_val<<16)+(in->b_val);
}
if(arena_mem_type==MEM_TYPE_FOUR)
{
mlarge=(struct arena_mem_norm*)out;
mlarge->processID=in->processID;
mlarge->opcode=(in->mod<<28)+(in->a_pref<<25)+(in->b_pref<<22)+in->
opcode;
mlarge->arg1=in->a_val;
mlarge->arg2=in->b_val;
}

}
void unpack(int ip,struct unpacked_op_mem *out)
{
struct array_mem_small *msmall;
struct array_mem_mid *mmid;
struct array_mem_norm *mlarge;

if(arena_mem_type==MEM_TYPE_ONE)
{
```

```
msmall=(struct mem_type_small*)arena;
out->b_val=(msmall[ip].mem>>1)&63;
out->a_val=(msmall[ip].mem>>7)&63;
out->b_pref=(msmall[ip].mem>>13)&7;
out->a_pref=(msmall[ip].mem>>16)&7;
out->mod=(msmall[ip].mem>>19)&15;
out->opcode=(msmall[ip].mem>>23)&31;
out->processID=(msmall[ip].mem>>28)&15;
}
if(arena_mem_type==MEM_TYPE_TWO)
{
mmid=(struct mem_type_mid*)arena;
out->b_val=(mmid[ip].arg1_arg2)&65535;
out->a_val=(mmid[ip].arg1_arg2>>16)&65535;
out->b_pref=(mmid[ip].processID_opcode>>2)&7;
out->a_pref=(mmid[ip].processID_opcode>>5)&7;
out->mod=(mmid[ip].processID_opcode>>8)&15;
out->opcode=(mmid[ip].processID_opcode>>12)&31;
out->processID=(mmid[ip].processID_opcode>>16)&65535;
}
if(arena_mem_type==MEM_TYPE_FOUR)
{
mlarge=(struct mem_type_norm*)arena;
out->a_val=mlarge[ip].arg1;
out->b_val=mlarge[ip].arg2;
out->processID=mlarge[ip].processID;
out->opcode=(mlarge[ip].opcode)&31;
out->b_pref=(mlarge[ip].opcode>>22)&7;
out->a_pref=(mlarge[ip].opcode>>25)&7;
out->mod=(mlarge[ip].opcode>>28)&15;
}
}
void pack2mem(int ip,struct unpacked_op_mem *in)
{
struct array_mem_small *msmall;
struct array_mem_mid *mmid;
struct array_mem_norm *mlarge;

if(arena_mem_type==MEM_TYPE_ONE)
{
msmall=(struct arena_mem_small*)arena;
msmall[ip].mem=(in->processID<<28)+(in->opcode<<23)+(in->mod<<19)+(in->
a_pref<<16)+(in->b_pref<<13)+(in->a_val<<7)+(in->b_val<<1);
}
if(arena_mem_type==MEM_TYPE_TWO)
```

```
{
mmid=(struct arena_mem_mid*)arena;
mmid[ip].processID_opcode=(in->processID<<16)+(in->opcode<<12)+(in->mod<
<8)+(in->a_pref<<5)+(in->b_pref<<2);
mmid[ip].arg1_arg2=(in->a_val<<16)+(in->b_val);
}
if(arena_mem_type==MEM_TYPE_FOUR)
{
mlarge=(struct arena_mem_norm*)arena;
mlarge[ip].processID=in->processID;
mlarge[ip].opcode=(in->mod<<28)+(in->a_pref<<25)+(in->b_pref<<22)+in->
opcode;
mlarge[ip].arg1=in->a_val;
mlarge[ip].arg2=in->b_val;
}

}
```

A.17 list_util.h

```
void die(char *s);
void add_vt(struct Process *proc,struct var_table *vt);
void add_proc(struct Process *proc);
void add_node(struct instruction_node *node,struct Process *proc);
void add_thread(struct process_thread *thread,struct process_task *task);
void add_task(struct process_task *task);
void del_task(struct process_task *task);
void addr2coords(int addr,int *x,int *y);
void get_p_attr(int ID,char *car,int *col);
void add_killed_task(struct process_task *task);
void add_thread_rev(struct process_thread *thread,struct process_task *task);
void ctout(struct process_thread *pt);
void cpout(struct process_task *ptask);
```

A.18 list_utils.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<gtk/gtk.h>
#include"main.h"
```

```
void die(char *s)
{
//char s[MAXCHAR];
sprintf(out_str,"%s\n",s);
fputs(out_str,fpout);
exit(1);
}
void add_vt(struct Process *proc,struct var_table *vt)
{
if(proc->pc->vt_first==NULL)
{
proc->pc->vt_first=vt;
proc->pc->vt_last=vt;
return;
}
vt->prev=proc->pc->vt_last;
proc->pc->vt_last->next=vt;
proc->pc->vt_last=vt;
}
void add_proc(struct Process *proc)
{
if(proc_primo==NULL)
{
proc_primo=proc;
proc_ultimo=proc;
}
else
{
proc->prev=proc_ultimo;
proc_ultimo->next=proc;
proc_ultimo=proc;
}
}
void add_node(struct instruction_node *node,struct Process *proc)
{
if(proc->pc->first==NULL)
{
proc->pc->first=node;
proc->pc->last=node;
return;
}
proc->pc->last->next=node;
node->prev=proc->pc->last;
proc->pc->last=node;
}
```

```
void add_thread(struct process_thread *thread,struct process_task *task)
{
thread->ptask=task;
if(task->primo_thread==NULL)
{
task->primo_thread=thread;
task->ultimo_thread=thread;
thread->prev=NULL;
thread->next=NULL;
return;
}
task->ultimo_thread->next=thread;
thread->prev=task->ultimo_thread;
thread->next=NULL;
task->ultimo_thread=thread;
}
void add_thread_rev(struct process_thread *thread,struct process_task *task)
{
thread->ptask=task;
if(task->primo_thread==NULL)
{
task->primo_thread=thread;
task->ultimo_thread=thread;
thread->prev=NULL;
thread->next=NULL;
return;
}
task->primo_thread->prev=thread;
thread->prev=NULL;
thread->next=task->primo_thread;
task->primo_thread=thread;
}
void add_task(struct process_task *task)
{
if(primo_task==NULL)
{
primo_task=task;
ultimo_task=task;
task->prev=NULL;
task->next=NULL;
return;
}
ultimo_task->next=task;
task->prev=ultimo_task;
task->next=NULL;
```

```
ultimo_task=task;
}
void del_task(struct process_task *task)
{
struct process_thread *pt,*opt;
if(primo_task==ultimo_task)
{
primo_task=NULL;
ultimo_task=NULL;
}
else
{
if(task->prev)
{task->prev->next=task->next;}else{primo_task=task->next;primo_task->
prev=NULL;}
if(task->next)
{task->next->prev=task->prev;}else{ultimo_task=task->prev;ultimo_task->
next=NULL;}
}
pt=task->primo_thread;
//delete all related threads
while(pt){
opt=pt;
pt=pt->next;
free(opt);
};
free(task);
}
void del_thread(struct process_thread *pt)
{
pt->ptask->n_threads--;
if(pt==pt->ptask->cur_thread)
{
pt->ptask->cur_thread=pt->next;
if(pt->next==NULL) pt->ptask->cur_thread=pt->ptask->primo_thread;
}
if(pt->ptask->primo_thread==pt->ptask->ultimo_thread)
{
pt->ptask->primo_thread=NULL;
pt->ptask->ultimo_thread=NULL;
pt->ptask->cur_thread=NULL;
}
else
{
if(pt->prev)
{pt->prev->next=pt->next;}else{pt->ptask->primo_thread=pt->next;pt->
ptask->primo_thread->prev=NULL;}
if(pt->next)
{pt->next->prev=pt->prev;}else{pt->ptask->ultimo_thread=pt->prev;pt->
ptask->ultimo_thread->next=NULL;}
}
free(pt);
}
void add_killed_task(struct process_task *task)
{
if(first_killed_task==NULL)
{
first_killed_task=task;
last_killed_task=task;
task->prev=NULL;
task->next=NULL;
return;
}
last_killed_task->next=task;
task->prev=last_killed_task;
task->next=NULL;
last_killed_task=task;
}
void addr2coords(int addr,int *x,int *y)
{
*x=addr % sc_x;
*y=(int)(((double)addr)/((double)sc_x));
}
void get_p_attr(int ID,char *car,int *col)
{
struct process_task *proc;
*car=63;
*col=-1;
if(ID==0) {*car='.';*col=0;return;}
for(proc=primo_task;proc;proc=proc->next)
{
if(proc->ID==ID)
{
*car=proc->out_symbol;
*col=proc->out_color;
return;
}
}
}
void ctout(struct process_thread *pt)
```

```
{
struct process_thread *ppt;
for(ppt=pt->ptask->primo_thread;ppt;ppt=ppt->next)
{
ppt->communication_in_a=pt->communication_out_a;
ppt->communication_in_b=pt->communication_out_b;
}
}
void cpout(struct process_task *ptask)
{
struct process_task *pptask;
for(pptask=primo_task;pptask;pptask=pptask->next)
{
pptask->communication_in_a=ptask->communication_out_a;
pptask->communication_in_b=ptask->communication_out_b;
}
}
```

A.19 execute.h

```
int execute (struct unpacked_op_mem *code,struct process_thread *pt);
```

A.20 execute.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<ctype.h>
#include<gtk/gtk.h>
#include"main.h"
#include"list_utils.h"
//#include"parse2.h"
#include"pack.h"
//#include"init_game.h"
//#include"scheduler.h"
#include"execute.h"
#include"debug_output.h"
#include"x11_output.h"

void get_I_field(struct unpacked_op_mem *IR,int addr_mode,int val, int IP,int
ID,int *RRPA)
{
```

```
int RPA,PIP;
struct unpacked_op_mem TMP;
if(addr_mode==op_IMM) {RPA=IP;*RRPA=RPA;unpack(RPA,IR);}
else
{
RPA=(IP+val)%size_arena;
if(addr_mode==op_B_IND_PREDEC)
{
unpack(RPA,&TMP);
(--TMP.b_val)%size_arena;
TMP.processID=ID;
pack2mem(RPA,&TMP);
RPA=(RPA+TMP.b_val)%size_arena;
}
if(addr_mode==op_A_IND_PREDEC)
{
unpack(RPA,&TMP);
(--TMP.a_val)%size_arena;
TMP.processID=ID;
pack2mem(RPA,&TMP);
RPA=(RPA+TMP.a_val)%size_arena;
}
if(addr_mode==op_B_IND)
{
unpack(RPA,&TMP);
RPA=(RPA+TMP.b_val)%size_arena;
}
if(addr_mode==op_A_IND)
{
unpack(RPA,&TMP);
RPA=(RPA+TMP.a_val)%size_arena;
}
if(addr_mode==op_B_IND_POSTINC)
{
unpack(RPA,&TMP);
PIP=RPA;
RPA=(RPA+TMP.b_val)%size_arena;
}
if(addr_mode==op_A_IND_POSTINC)
{
unpack(RPA,&TMP);
PIP=RPA;
RPA=(RPA+TMP.a_val)%size_arena;
}
unpack(RPA,IR);
```

```
*RRPA=RPA;
if(addr_mode==op_B_IND_POSTINC)
{
TMP.processID=ID;
(++TMP.b_val)%size_arena;
pack2mem(PIP,&TMP);
}
if(addr_mode==op_A_IND_POSTINC)
{
TMP.processID=ID;
(++TMP.a_val)%size_arena;
pack2mem(PIP,&TMP);
}
}

}
int execute (struct unpacked_op_mem *code,struct process_thread *pt)
{
int alive,RPA,RPB;
struct unpacked_op_mem IRA,IRB;
struct process_task *ptask;
struct process_thread *new_thread;
alive=ALIVE;
code->processID=pt->ptask->ID;
get_I_field(&IRA,code->a_pref,code->a_val,pt->IP,code->processID,&RPA);
get_I_field(&IRB,code->b_pref,code->b_val,pt->IP,code->processID,&RPB);
print_ex_data(code,&IRA,&IRB,RPA,RPB,pt);
if(vo_mode>=VO_FRAMEBUFFER)
{
cell_refresh(pt->IP,pt);
}
if(output_mode==OUTPUT_DEBUG2 && (vo_mode==VO_NONE || b_log))
{
print_debug3();
}
switch(code->opcode)
{
case op_NOP:
pt->IP=(pt->IP+1)%size_arena;
break;
case op_DAT:
alive=DEAD;
break;
case op_MOV:
switch(code->mod)
```

```
{
case op_A:
IRB.a_val=IRA.a_val;
IRB.processID=pt->ptask->ID;
pack2mem(RPB,&IRB);
break;
case op_B:
IRB.b_val=IRA.b_val;
IRB.processID=pt->ptask->ID;
pack2mem(RPB,&IRB);
break;
case op_AB:
IRB.b_val=IRA.a_val;
IRB.processID=pt->ptask->ID;
pack2mem(RPB,&IRB);
break;
case op_BA:
IRB.a_val=IRA.b_val;
IRB.processID=pt->ptask->ID;
pack2mem(RPB,&IRB);
break;
case op_F:
IRB.a_val=IRA.a_val;
IRB.b_val=IRA.b_val;
IRB.processID=pt->ptask->ID;
pack2mem(RPB,&IRB);
break;
case op_X:
IRB.a_val=IRA.b_val;
IRB.b_val=IRA.a_val;
IRB.processID=pt->ptask->ID;
pack2mem(RPB,&IRB);
break;
case op_I:
IRA.processID=pt->ptask->ID;
pack2mem(RPB,&IRA);
break;
default:
break;
}
pt->IP=(pt->IP+1)%size_arena;
if(vo_mode>=VO_FRAMEBUFFER)
{
cell_refresh(RPB,pt);
}
```

```
break;
case op_ADD:
IRB.processID=pt->ptask->ID;
switch(code->mod)
{
case op_A:
IRB.a_val=(IRA.a_val+IRB.a_val)%size_arena;
break;
case op_B:
IRB.b_val=(IRA.b_val+IRB.b_val)%size_arena;
break;
case op_AB:
IRB.b_val=(IRA.a_val+IRB.b_val)%size_arena;
break;
case op_BA:
IRB.a_val=(IRA.b_val+IRB.a_val)%size_arena;
break;
case op_I:
case op_F:
IRB.a_val=(IRA.a_val+IRB.a_val)%size_arena;
IRB.b_val=(IRA.b_val+IRB.b_val)%size_arena;
break;
case op_X:
IRB.a_val=(IRA.b_val+IRB.b_val)%size_arena;
IRB.b_val=(IRA.a_val+IRB.a_val)%size_arena;
break;
default:
break;
}
pack2mem(RPB,&IRB);
pt->IP=(pt->IP+1)%size_arena;
if(vo_mode>=VO_FRAMEBUFFER)
{
cell_refresh(RPB,pt);
}
break;
case op_SUB:
IRB.processID=pt->ptask->ID;
switch(code->mod)
{
case op_A:
IRB.a_val=(IRB.a_val-IRA.a_val)%size_arena; //or maybe IRA-IRB ?!?!?to check
break;
case op_B:
IRB.b_val=(IRB.b_val-IRA.b_val)%size_arena;
```

```
break;
case op_AB:
IRB.b_val=(IRB.b_val-IRA.a_val)%size_arena;
break;
case op_BA:
IRB.a_val=(IRB.a_val-IRA.b_val)%size_arena;
break;
case op_I:
case op_F:
IRB.a_val=(IRB.a_val-IRA.a_val)%size_arena;
IRB.b_val=(IRB.b_val-IRA.b_val)%size_arena;
break;
case op_X:
IRB.a_val=(IRB.b_val-IRA.b_val)%size_arena;
IRB.b_val=(IRB.a_val-IRA.a_val)&size_arena;
break;
default:
break;
}
pack2mem(RPB,&IRB);
pt->IP=(pt->IP+1)%size_arena;
if(vo_mode>=VO_FRAMEBUFFER)
{
cell_refresh(RPB,pt);
}
break;
case op_MUL:
IRB.processID=pt->ptask->ID;
switch(code->mod)
{
case op_A:
IRB.a_val=(IRA.a_val*IRB.a_val)%size_arena;
break;
case op_B:
IRB.b_val=(IRA.b_val*IRB.b_val)%size_arena;
break;
case op_AB:
IRB.b_val=(IRA.a_val*IRB.b_val)%size_arena;
break;
case op_BA:
IRB.a_val=(IRA.b_val*IRB.a_val)%size_arena;
break;
case op_I:
case op_F:
IRB.a_val=(IRA.a_val*IRB.a_val)%size_arena;
IRB.b_val=(IRA.b_val*IRB.b_val)%size_arena;
break;
case op_X:
IRB.a_val=(IRA.b_val*IRB.b_val)%size_arena;
IRB.b_val=(IRA.a_val*IRB.a_val)%size_arena;
break;
default:
break;
}
pack2mem(RPB,&IRB);
pt->IP=(pt->IP+1)%size_arena;
if(vo_mode>=VO_FRAMEBUFFER)
{
cell_refresh(RPB,pt);
}
break;
case op_DIV:
IRB.processID=pt->ptask->ID;
switch(code->mod)
{
case op_A:
if(IRA.a_val==0) {alive=DEAD;} else
{IRB.a_val=(IRB.a_val/IRA.a_val)%size_arena;}
break;
case op_B:
if(IRA.b_val==0) {alive=DEAD;} else
{IRB.b_val=(IRB.b_val/IRA.b_val)%size_arena;}
break;
case op_AB:
if(IRA.a_val==0) {alive=DEAD;} else
{IRB.b_val=(IRB.b_val/IRA.a_val)%size_arena;}
break;
case op_BA:
if(IRA.b_val==0) {alive=DEAD;} else
{IRB.a_val=(IRB.a_val/IRA.b_val)%size_arena;}
break;
case op_F:
case op_I:
if((IRA.a_val==0)||(IRA.b_val==0)) {alive=DEAD;} else
{
IRB.a_val=(IRB.a_val/IRA.a_val)%size_arena;
IRB.b_val=(IRB.b_val/IRA.b_val)%size_arena;
}
break;
case op_X:
```

```
if((IRA.a_val==0)||(IRA.b_val==0)) {alive=DEAD;} else
{
IRB.a_val=(IRB.b_val/IRA.b_val)%size_arena;
IRB.b_val=(IRB.a_val/IRA.a_val)%size_arena;
}
break;
default:
break;
}
pack2mem(RPB,&IRB);
pt->IP=(pt->IP+1)%size_arena;
if(vo_mode>=VO_FRAMEBUFFER)
{
cell_refresh(RPB,pt);
}
break;
case op_MOD:
IRB.processID=pt->ptask->ID;
switch(code->mod)
{
case op_A:
if(IRA.a_val==0) {alive=DEAD;} else
{IRB.a_val=(IRB.a_val%IRA.a_val)%size_arena;}
break;
case op_B:
if(IRA.b_val==0) {alive=DEAD;} else
{IRB.b_val=(IRB.b_val%IRA.b_val)%size_arena;}
break;
case op_AB:
if(IRA.a_val==0) {alive=DEAD;} else
{IRB.b_val=(IRB.b_val%IRA.a_val)%size_arena;}
break;
case op_BA:
if(IRA.b_val==0) {alive=DEAD;} else
{IRB.a_val=(IRB.a_val%IRA.b_val)%size_arena;}
break;
case op_F:
case op_I:
if((IRA.a_val==0)||(IRA.b_val==0)) {alive=DEAD;} else
{
IRB.a_val=(IRB.a_val%IRA.a_val)%size_arena;
IRB.b_val=(IRB.b_val%IRA.b_val)%size_arena;
}
break;
case op_X:
```

```
if((IRA.a_val==0)||(IRA.b_val==0)) {alive=DEAD;} else
{
IRB.a_val=(IRB.b_val%IRA.b_val)%size_arena;
IRB.b_val=(IRB.a_val%IRA.a_val)%size_arena;
}
break;
default:
break;
}
pack2mem(RPB,&IRB);
pt->IP=(pt->IP+1)%size_arena;
if(vo_mode>=VO_FRAMEBUFFER)
{
cell_refresh(RPB,pt);
}
break;
case op_JMP:
pt->IP=RPA;
break;
case op_JMZ:
switch(code->mod)
{
case op_A:
case op_BA:
if(IRB.a_val==0) {pt->IP=RPA;} else {pt->IP=(pt->IP+1)%size_arena;}
break;
case op_B:
case op_AB:
if(IRB.b_val==0) {pt->IP=RPA;} else {pt->IP=(pt->IP+1)%size_arena;}
break;
case op_F:
case op_X:
case op_I:
if((IRB.a_val==0)&&(IRB.b_val==0))
{ pt->IP=RPA;} else {pt->IP=(pt->IP+1)%size_arena;}
break;
default:
break;
}
break;
case op_JMN:
switch(code->mod)
{
case op_A:
case op_BA:
```

```
if(IRB.a_val!=0) {pt->IP=RPA;} else {pt->IP=(pt->IP+1)%size_arena;}
break;
case op_B:
case op_AB:
if(IRB.b_val!=0) {pt->IP=RPA;} else {pt->IP=(pt->IP+1)%size_arena;}
break;
case op_F:
case op_X:
case op_I:
if((IRB.a_val!=0)&&(IRB.b_val!=0))
{ pt->IP=RPA;} else {(pt->IP++)%size_arena;}
break;
default:
break;
}
break;
case op_DJN:
IRB.processID=pt->ptask->ID;
switch(code->mod)
{
case op_A:
case op_BA:
(IRB.a_val--)%size_arena;
if(IRB.a_val!=0) {pt->IP=RPA;}else{pt->IP=(pt->IP+1)%size_arena;}
break;
case op_B:
case op_AB:
(IRB.b_val--)%size_arena;
if(IRB.b_val!=0) {pt->IP=RPA;}else{pt->IP=(pt->IP+1)%size_arena;}
break;
case op_F:
case op_I:
case op_X:
(IRB.a_val--)%size_arena;
(IRB.b_val--)%size_arena;
if((IRB.a_val!=0)||(IRB.b_val!=0))
{pt->IP=RPA;}else{(pt->IP++)%size_arena;}
break;
default:
break;
}
pack2mem(RPB,&IRB);
break;
case op_CMP:
switch(code->mod)
```

```
{
case op_A:
if(IRA.a_val==IRB.a_val)
{pt->IP=(pt->IP+2)%size_arena;}
else{pt->IP=(pt->IP+1)%size_arena;}
break;
case op_B:
if(IRA.a_val==IRB.a_val)
{pt->IP=(pt->IP+2)%size_arena;}
else{pt->IP=(pt->IP+1)%size_arena;}
break;
case op_AB:
if(IRA.a_val==IRB.b_val)
{pt->IP=(pt->IP+2)%size_arena;}
else{pt->IP=(pt->IP+1)%size_arena;}
break;
case op_BA:
if(IRA.b_val==IRB.a_val)
{pt->IP=(pt->IP+2)%size_arena;}
else{pt->IP=(pt->IP+1)%size_arena;}
break;
case op_F:
if((IRA.a_val==IRB.a_val)&&(IRA.b_val==IRB.b_val))
{pt->IP=(pt->IP+2)%size_arena;}
else{pt->IP=(pt->IP+1)%size_arena;}
break;
case op_X:
if((IRA.a_val==IRB.b_val)&&(IRA.b_val==IRB.a_val))
{pt->IP=(pt->IP+2)%size_arena;}
else{pt->IP=(pt->IP+1)%size_arena;}
break;
case op_I:
if((IRA.opcode==IRB.opcode)&&
(IRA.mod==IRB.mod)&&
(IRA.a_pref==IRB.b_pref)&&
(IRA.a_val==IRB.a_val)&&
(IRA.b_pref==IRB.b_pref)&&
(IRA.b_val==IRB.b_val))
{pt->IP=(pt->IP+2)%size_arena;}
else{pt->IP=(pt->IP+1)%size_arena;}
break;
default:
break;
}
break;
```

```
case op_SLT:
switch(code->mod)
{
case op_A:
if(IRA.a_val<IRB.a_val)
{pt->IP=(pt->IP+2)%size_arena;}
else {pt->IP=(pt->IP+1)%size_arena;}
break;
case op_B:
if(IRA.b_val<IRB.b_val)
{pt->IP=(pt->IP+2)%size_arena;}
else {pt->IP=(pt->IP+1)%size_arena;}
break;
case op_AB:
if(IRA.a_val<IRB.b_val)
{pt->IP=(pt->IP+2)%size_arena;}
else {pt->IP=(pt->IP+1)%size_arena;}
break;
case op_BA:
if(IRA.b_val<IRB.a_val)
{pt->IP=(pt->IP+2)%size_arena;}
else {pt->IP=(pt->IP+1)%size_arena;}
break;
case op_F:
case op_I:
if((IRA.a_val<IRB.a_val)&&(IRA.b_val<IRB.b_val))
{pt->IP=(pt->IP+2)%size_arena;}
else {pt->IP=(pt->IP+1)%size_arena;}
break;
case op_X:
if((IRA.a_val<IRB.b_val)&&(IRA.b_val<IRB.a_val))
{pt->IP=(pt->IP+2)%size_arena;}
else {pt->IP=(pt->IP+1)%size_arena;}
break;
default:
break;
}
break;
case op_SPL:
//create new thread
ptask=pt->ptask;
if(ptask->n_threads<maxprocesses)
{
new_thread=(struct process_thread*)malloc(sizeof(struct process_thread));
if(new_thread==NULL) die("error creating the new thread");
ptask->n_threads++;
new_thread->ptask=ptask;
new_thread->prev=NULL;
new_thread->next=NULL;
new_thread->communication_in_a=0;
new_thread->communication_out_a=0;
new_thread->communication_in_b=0;
new_thread->communication_out_b=0;
new_thread->IP=RPA;
// and resume to next instruction
pt->IP=(pt->IP+1)%size_arena;
// update gtk_view
if(vo_mode==VO_X11) gtk_update_warrior(ptask);
//set new process to go last in process task's queue (repeat the father first)
add_thread_rev(new_thread,ptask);
ptask->cur_thread=ptask->cur_thread->prev;
if(ptask->cur_thread==NULL) ptask->cur_thread=ptask->ultimo_thread;
}
break;
case op_CTOUT:
switch(code->mod)
{
case op_A:
pt->communication_out_a=IRA.a_val;
pt->communication_out_b=IRB.a_val;
break;
case op_B:
pt->communication_out_a=IRA.b_val;
pt->communication_out_b=IRB.b_val;
break;
case op_AB:
case op_F:
case op_I:
pt->communication_out_a=IRA.a_val;
pt->communication_out_b=IRB.b_val;
break;
case op_BA:
case op_X:
pt->communication_out_a=IRA.b_val;
pt->communication_out_b=IRB.a_val;
break;

default:
break;
}
```

```
ctout(pt);
pt->IP=(pt->IP+1)%size_arena;
break;
case op_CTIN:
switch(code->mod)
{
case op_A:
IRA.a_val=pt->communication_out_a;
IRB.a_val=pt->communication_out_b;
break;
case op_B:
IRA.b_val=pt->communication_out_a;
IRB.b_val=pt->communication_out_b;
break;
case op_AB:
case op_F:
case op_I:
IRA.a_val=pt->communication_out_a;
IRB.b_val=pt->communication_out_b;
break;
case op_BA:
case op_X:
IRA.b_val=pt->communication_out_a;
IRB.a_val=pt->communication_out_b;
break;

default:
break;
}
pt->IP=(pt->IP+1)%size_arena;
if(vo_mode>=VO_FRAMEBUFFER)
{
cell_refresh(RPB,pt);
cell_refresh(RPA,pt);
}
break;
case op_CPOUT:
switch(code->mod)
{
case op_A:
pt->ptask->communication_out_a=IRA.a_val;
pt->ptask->communication_out_b=IRB.a_val;
break;
case op_B:
pt->ptask->communication_out_a=IRA.b_val;
```

```
pt->ptask->communication_out_b=IRB.b_val;
break;
case op_AB:
case op_F:
case op_I:
pt->ptask->communication_out_a=IRA.a_val;
pt->ptask->communication_out_b=IRB.b_val;
break;
case op_BA:
case op_X:
pt->ptask->communication_out_a=IRA.b_val;
pt->ptask->communication_out_b=IRB.a_val;
break;

default:
break;
}
cpout(pt->ptask);
pt->IP=(pt->IP+1)%size_arena;
break;
case op_CPIN:
switch(code->mod)
{
case op_A:
IRA.a_val=pt->ptask->communication_out_a;
IRB.a_val=pt->ptask->communication_out_b;
break;
case op_B:
IRA.b_val=pt->ptask->communication_out_a;
IRB.b_val=pt->ptask->communication_out_b;
break;
case op_AB:
case op_F:
case op_I:
IRA.a_val=pt->ptask->communication_out_a;
IRB.b_val=pt->ptask->communication_out_b;
break;
case op_BA:
case op_X:
IRA.b_val=pt->ptask->communication_out_a;
IRB.a_val=pt->ptask->communication_out_b;
break;

default:
break;
```

```
}
pt->IP=(pt->IP+1)%size_arena;
if(vo_mode>=VO_FRAMEBUFFER)
{
cell_refresh(RPB,pt);
cell_refresh(RPA,pt);
}
break;
default:
break;
}
// if(output_mode>=OUTPUT_DEBUG)
// {
// sprintf(out_str,"=>newIP=%d\n",pt->IP);
// if(vo_mode==VO_NONE && log_mode) fputs(out_str,fpout);
// //if(vo_mode==VO_FRAMEBUFFER);
// }
return alive;
}
```

A.21 debug_output.h

```
void print_ex_data(struct unpacked_op_mem *code,struct unpacked_op_mem
*IRA,struct unpacked_op_mem *IRB,int RPA,int RPB,struct process_thread *pt);
void cell_refresh(int addr,struct process_thread *pt);
void print_debug3(void);
```

A.22 debug_output.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<ctype.h>
#include<gtk/gtk.h>
#include"main.h"
#include"list_utils.h"
//#include"parse2.h"
#include"pack.h"
//#include"init_game.h"
//#include"scheduler.h"
#include"execute.h"
#include"debug_output.h"
```

```
#include"txt_output.h"
#include"x11_output.h"

extern void gtk_display_curr_instr(char *ss);

void get_str(struct unpacked_op_mem *code,char *m_op,char *m_apref,char
*m_bpref,char *m_mod)
{
static char *op_table[]={
"nop","1","dat","0","mov","2","add","3","sub","4","mul","5","div","6","mod",
"7","jmp","8",
"jmz","9","jmn","10","djn","11","spl","12","cmp","13","seq","14","sne","15",
"slt","16","ldp","17",
"stp","18","NULL","-1"
};
static char *pref_table[]={
"#","0","$","2","*","3","@","4","{","5","<","6","}","7",">","1","NULL","-1"
};
static char *mod_table[]={
"a","0","b","1","ab","2","ba","3","f","4","x","5","i","6","NULL","-1"
};

int n,rv;
for(n=1;;n+=2)
{
rv=atoi(op_table[n]);
if(rv==code->opcode) {strcpy(m_op,op_table[n-1]);break;}
}
for(n=1;;n+=2)
{
rv=atoi(pref_table[n]);
if(rv==code->a_pref) {strcpy(m_apref,pref_table[n-1]);break;}
}
for(n=1;;n+=2)
{
rv=atoi(pref_table[n]);
if(rv==code->b_pref) {strcpy(m_bpref,pref_table[n-1]);break;}
}
for(n=1;;n+=2)
{
rv=atoi(mod_table[n]);
if(rv==code->mod) {strcpy(m_mod,mod_table[n-1]);break;}
}
}
void print_ex_data(struct unpacked_op_mem *code,struct unpacked_op_mem
*IRA,struct unpacked_op_mem *IRB,int RPA,int RPB,struct process_thread *pt)
{
char m_op[8],m_apref[5],m_bpref[5],m_mod[5];
int m_aval=0,m_bval=0;
if(output_mode>=OUTPUT_DEBUG2)
{
get_str(code,&m_op[0],&m_apref[0],&m_bpref[0],&m_mod[0]);
sprintf(out_str,"code:{proc=%d| %s %s %d,%s %d |IP=%d}
",code->processID,m_op,m_apref,code->a_val,m_bpref,code->b_val,pt->IP);
if((vo_mode==VO_NONE) /*&& (log_mode)*/)
{
fputs(out_str,fpout);
fputs("\n",fpout);
}
if(vo_mode==VO_FRAMEBUFFER) mvaddstr(sc_y+1+yd,0,out_str);
}
if(vo_mode==VO_FRAMEBUFFER)
{
get_str(code,&m_op[0],&m_apref[0],&m_bpref[0],&m_mod[0]);
sprintf(out_str,"code:{proc=%d| %s %s %d,%s %d |IP=%d}
",code->processID,m_op,m_apref,code->a_val,m_bpref,code->b_val,pt->IP);
mvaddstr(sc_y+1+yd,0,out_str);
}
if(vo_mode==VO_X11)
{
get_str(code,&m_op[0],&m_apref[0],&m_bpref[0],&m_mod[0]);
sprintf(out_str,"code:{proc=%d| %s %s %d,%s %d |IP=%d}
",code->processID,m_op,m_apref,code->a_val,m_bpref,code->b_val,pt->IP);
gtk_display_curr_instr(out_str);
}
}
void cell_refresh(int addr,struct process_thread *pt)
{
if(vo_mode==VO_FRAMEBUFFER) curses_cell_refresh(addr,pt);
if(vo_mode==VO_X11) x11_cell_refresh(addr,pt);
}
void print_debug3()
{
int count,x_pos,y_pos,col;
char car;
struct unpacked_op_mem mem;
for(count=0;count<size_arena;count++)
{
for(x_pos=0;x_pos<max_x;x_pos++)
{
```

```
if(++count>size_arena) break;
unpack(count,&mem);
get_p_attr(mem.processID,&car,&col);
sprintf(out_str,"%c",car);
fputs(out_str,fpout);
}
sprintf(out_str,"\n");
fputs(out_str,fpout);
}
sprintf(out_str,"\n");
fputs(out_str,fpout);
}
```

Appendix B

Terminology of Malicious Programs

B.1 Viruses

A virus is a program that is able to infect other programs by modifying them to include a possibly evolved copy of itself. A virus needs a host program to infect. A virus attacks a non-running copy of the program (the image on the disk).

B.2 Worms

Worms are network viruses, primarily replicating on networks. Usually a worm will execute itself automatically on a remote machine without any extra help from a user. Worms attack a running copy of a program. Worms are typically standalone applications without a host program.

B.2.1 Mailer and Mass-Media Worms

Mailers and mass-mailer worms comprise a special class of computer worms, which send themselves in an e-mail.

B.2.2 Octopus

An octopus is a sophisticated kind of computer worm that exists as a set of programs on more than one computer on a network. For example, head and tail copies are installed on individual computers that communicate with each other to perform a function.

B.2.3 Rabbits

A rabbit is a special computer worm that exists as a single copy of itself at any point in time as it "jumps around" on the network hosts. Other researchers use the term rabbit to describe crafty, malicious applications that usually run themselves recursively to fill memory with their own copies and to slow down processing time consuming CPU time. Such malicious code uses too much memory and thus can cause serious side effects an a machine within other applications that are not prepared to work under low-memory conditions and that unexpectedly cease functioning.

B.3 Logic Bombs

A logic bomb is a programmed malfunction of a legitimate application. An application, for example, might delete itself from the disk after a couple of runs as a copy protection scheme; a programmer might want to include some extra code to perform a malicious action an a certain condition or on certain systems when the application is used.

B.4 Trojan Horses

Perhaps the simpliest kind of malicious program is a Trojan horse. Trojan horses try to appeal to and interest the user with some useful functionality to entice the user to run the program. At run-time the program executes the declared functionality but also some other hidden and malicious task.

B.4.1 Backdoors

A backdoor is a malicious hacker's tool of choice that allows remote connections to system. Usually backdoors bypasses all security systems and give a direct access to a system. A typical backdoor opens a network port (UDP/TCP) on the host when it is executed. Then, the listening backdoor waits for a remote connection from the attacker and allows the attacker to connect to the system.

B.5 Germs

Germs are first-generation viruses in a form that the virus cannot generate to its usual infection process. Usually, when the virus is compiled for the first time, it exists in a special form (is not a result of an infection) and does not have a host program attached to it. Germs will not have the usual marks that most viruses use in the second-generation form to flag infected files to avoid reinfecting an already infected object. A germ of an encrypted or polymorphic virus is usually not encrypted but is plain, readable code. Detecting Germs might need to be done differently from detecting second, and later,-generation infections.

B.6 Exploits

Exploit code is specific to a single vulnerability or a set of vulnerabilities. Its goal is to run a program on a (possibly remote, networked) system automatically or provide some other form of more highly privileged access to the target system.

B.7 Downloaders

A downloader is yet another malicious program that installs a set of other items on a machine that is under attack. Usually, a downloader is sent in e-mail, and when it is executed, it downloads malicious content from Web sites or other location and then extracts and runs its content.

B.8 Dialers

Dialers got their relatively early start during the heyday of dial-up connections to bullettin board systems in homes. The concept driving a dialer is to make money for the people behind the dialer by having its user call via premium-rate phone numbers. Thus, the person who runs the dialer might know the intent of the application, but the user is not aware of the charges.

B.9 Droppers

The original term refers to an "installer" for first-generation virus code. For example, boot viruses that firs exist as compiled files in binary form are often installed in the boot sector of a floppy using a dropper. The dropper writes the germ code to the boot sector of the diskette. Then the virus replicate on its own without ever generating the dropper form again.

B.10 Injectors

Injectors are special kind of droppers that usually install virus code in memory. An injector can be used to inject virus code in an active form on a disk interrupt handler. Then, the first time a user accesses a diskette, the virus begins to replicate itself normally. A special kind of injector is the network injector. Attackers also can use legitimate utilities, such as NetCat, to inject code into the network. Injectors are often used in a process called *seeding*. Seeding is a process that is used to inject virus code to several remote systems to cause an initial outbreak that is large enough to cause a quick epidemic.

B.11 Kits (Virus Generators)

Virus writers developed kits, such as the Virus Creation Laboratory (VCL), to generate new computer viruses automatically, using a menu-based application. With such tools, even novice users were able to develop harmful computer viruses without too much background knowledge. The Dark Avanger's virus mutation engine (MtE) is also able to generate polymorphics viruses.

B.12 Spammer Programs

Spammer programs are used to send unsolicited messages to Instant Messaging groups, newsgroups, or many kind of mobile device in form of e-mail or cell phone SMS messages. The primary motivation of spammers is to make money by generating traffic to Web sites. In addition, spam messages are often used to implement phishing attacks. For example, you might receive an e-mail message asking you to visit your bank's Web site and telling you that if you don't, they will disable your account. There is a link in the e-mail, however, that forwards you to the fraudster. If you fall victim to the attack, you might disclose personal information to the attacker on a silver plate. The fraudster wants to get your credit card number, account number, password, PIN, and other personal information to make money. In addition, you might become the prime subject of an identity theft as well.

B.13 Flooders

Malicious hackers use flooders to attack networked computer systems with an extra load of network traffic to carry out a denial of service (DoS) attack. When the Dos attack is performed simultaneously from many compromised systems (so-called zombie machines), the attack is called a distributed denial of service (DDoS) attack. Of course, there are much more sophisticated DoS attacks including SYN flood, packet fragmentation attacks, and other (mis-)sequencing attacks, traffic amplification, or traffic deflection, just to name the most common types.

B.14 Keyloggers

A keylogger captures on a compromised system, collecting sensitive information for the attacker. Such sensitive information might include names, passwords, PINs, birthdays, Social Security numbers, or credit card numbers.

B.15 Rootkits

Rootkits are a special set of malicious hacker tools that are used after the attacker has broken into a computer system and gained root-level access. Rootkits are used to hide an attack and the system alteration after an attack. Usually attackers break into a system with exploits and install modified versions of common tools. Such rootkits are called user-mode rootkits because the Trojanized application runs in user mode. Modern rootkits corrupt directly the kernel.

B.16 Joke Programs

Joke programs are not malicious, as Alan Solomon (author of one of the most widely used scanning engines today) once mentioned, "Whether a program should be classified as a joke program or as a Trojan largely depends on the sense of humor of the victim". Joke programs change or interrupt the normal behavior of your computer, creating a general distraction or nuisance. Such programs can be considered harmful in some sense. Consider, for example, a joke program that locks the system but never unlocks it. Thus, computers cannot be stopped safely. As a result, important data could be lost because it was never saved to the disk. Or worse, thee file allocation table could get corrupted, and the machine would become unbootable.

Bibliography

[SHLLCD] Kris Kaspersky, *ShellCoder's Programming Uncovered,* **Alist 2005**

[DBGG] Kris Kaspersky, *Hacker Debugging Uncovered,* **Alist 2005**

[AHO] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques,and Tools,* **Addison Wesley 1988**

[BOF] James C. Foster, *Buffer Overflow Attacks,* **Syngress 2005**

[VIR] Matteo Salin, *I Virus dei Computer,* **Livana Editrice 1989**

[SZOR] Peter Szor, *Virus Research and Defense,* **Addison Wesley 2005**

[NEUMNN] J. von Neunamm, A.W.Burks, *Theory of self-reproducing automata,* **University of Illinois Press 1966**

[LUDWG] M. Ludwig, *The Little Black Book of Computer Viruses,* **American Eagle Publications 1998**

[LDWAI] M. Ludwig, *Computer Viruses as Artificial Life,* **American Eagle Publications 1998**

[WNSKL] G. Winskel, *The Formal Semantic of Programming Languages,* **The MIT Press 1993**

[ADP] A. Pettorossi, *Theory of Computation vol.I,II,III,IV,* **Aracne 2002**

[PRPL] A. Pettorossi, *Elements of computability, decidability and complexity,* **Aracne 2006**

[LINSHLL] Steve Hanna, *Shellcoding for Linux and Windows Tutorial,* INTERNET ADDRESS: http://www.vividmachines.com/shellcode/shellcode.html

[WINSC] The Metasploit Project, *Windows System Call Table (NT,2000,XP,2003,Vista),* INTERNET ADDRESS: http://www.metasploit.com/users/opcode/syscalls.html

[LINASM] Derick Swanepoel, *Linux Assembly Tutorial,* INTERNET ADDRESS: http://www.fizik.itu.edu.tr/turhan/asm/mnasm.html

[LINASMELF] LiTle VxW, *Asm Tutorial for Linux and Elf File Format,* INTERNET ADDRESS: http://www.woodmann.com/0xf001/filez/29A-8.015.txt

[ELFFMT] Brian Raiter, *Executable and Linkable Format (ELF),* INTERNET ADDRESS: http://www.muppetlabs.com/ breadbox/software/ELF.txt

[APPL] Apple Inc., *Mac Os X ABI Mach-O File Format Reference,* INTERNET ADDRESS: http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/Reference

[VIRTHRY] Leonard Adleman, *An Abstract Theory of Computer Viruses,* INTERNET ADDRESS: http://vx.netlux.org/lib/ala01.html

[CPTVIR] Guillaume Bonfante, Matthieu Kaczmarek, Jean-Yves Mario, *Toward an Abstract Computer Virology,* INTERNET ADDRESS: http://vx.netlux.org/lib/agb00.html

[COHEN] Fred Cohen, *Computer Viruses- Theory and Experiments,* INTERNET ADDRESS: http://vx.netlux.org/lib/afc01.html

[CRWR] Stephen Beitzel, *Annotated Draft of the Proposed 1994 Core War Standard,* INTERNET ADDRESS: http://www.koth.org/info/icws94.html

[REDREF] ICWS'94 draft, *RedCode Reference,* INTERNET ADDRESS: http://www.koth.org/info/pmars-redcode-94.txt

[REDCDE] Ilmari Karonen, *The beginner's guide to Redcode,* INTERNET ADDRESS: http://vyznev.net/corewar/guide.html