IJRIT International Journal of Research in Information Technology, Volume 1, Issue 11, November, 2013, Pg. 201-210

International Journal of Research in Information Technology (IJRIT)

www.ijrit.com ISSN 2001-5569

Terminate and Stay Resident Viruses

Priyanka Mogha[1], Nitika Sharma[2], and Satish Sharma[3]

[1]Student, Dept. of Information Technology, Dronacharya College of Engineering
Gurgaon, Haryana, India
mogha.virgo20@gmail.com

[2]Student, Dept. of Information Technology, Dronacharya College of Engineering
Gurgaon, Haryana, India
nitika.3092@gmail.com

[3]Student, Dept. of Information Technology, Dronacharya College of Engineering
Gurgaon, Haryana, India
satishsharma1405@gmail.com

Abstract

Terminate and Stay Resident (TSR) is a computer system call in DOS computer operating systems that returns control to the system as if the program has quit, but keeps the program in memory, to be revived later by a hardware or software interrupt. These are also considered memory resident. In the computer world a program can install themselves in memory, and this part can remain active after the program has ended. This memory resident part is called a resident extension. Many viruses install themselves as resident extensions. Being memory resident provides several advantages for viruses. This would account for the fact that the majority of viruses stay in the memory. They will reside there until something external occurs like inserting a disk or executing a program. By being in the memory, a resident virus can do anything the operating system can do. Some will modify or damage the computer's software to hide in the memory. Stealth viruses are always memory resident.

***Keywords*:** *interrupts, DOS, memory resident, stealth viruses*

1. Introduction

The term *computer virus* is derived from and is in some sense analogous to a biological virus. The word *virus* itself is Latin for *poison*. Simplistically, biological viral infections are spread by the virus (a small shell containing genetic material) injecting its contents into a far larger organism's cell. The cell then is infected and converted into a biological factory producing replicants of the virus.
Similarly, a computer virus is a segment of machine code (typically 200-4000 bytes) that will copy itself (or a modified version of itself) into one or more larger "host" programs when it is activated. When these infected programs are run, the viral code is executed and the virus spreads further. Sometimes, what constitutes "programs" is more than simply applications: boot code, device drivers, and command interpreters also can be infected. Computer viruses cannot spread by infecting pure data; pure data files are not executed.

However, some data, such as files with spreadsheet input or text files for editing, may be interpreted by application programs.

Memory-resident viruses (also called **TSR** for *Terminate and Stay Resident*) load in the computer's RAM in order to infect executable files opened by the user. Non-resident viruses, once run, infect programs found on the hard drive.

The effects of a virus may range from simply displaying a ping-pong ball ricocheting across the screen to wiping out data, which is the most destructive kind of virus there is. As there is a broad range of viruses with widely varied effects, viruses are not classified based on what kind of damage they do, but on how they spread and infect computers. For this reason, there are different types of viruses:

- Worms are viruses which can spread over a network
- Trojan horses (trojans) are viruses which create a security hole in the computer (generally for their designer to gain entry to the infect system and take control of it)
- Logic bombs are viruses which can trigger on a specific event (like the system's date, or remote activation).

A new phenomenon has appeared in the past few years, that of hoaxes, i.e. notices received by e-mail(for example a report on the appearance of a new destructive virus or a chance to win a free mobile phone) along with a note telling the recipient to forward the message to everyone he or she knows. The purpose of this is to clog network traffic and spread misinformation.

2. Detecting a virus

Viruses reproduce by infecting "*host applications*," meaning that they copy a portion of executable code into an existing program. So to ensure that they work as planned, viruses are programmed to not infect the same file multiple times. To do so, they include a series of bytes in the infected application, to check if has already been infected: This is called a **virus signature**.

Antivirus programs rely on this signature, which is unique to each virus, in order to detect them. This method is called **signature scanning**, the oldest method used by antivirus software.
This method is only reliable if the antivirus program's virus database is up-to-date and includes signatures for all known viruses. However, this method cannot detect viruses which have not been archived by the publishers of the antivirus software. What's more, virus programmers have often given them camouflage features, making their signature hard to detect, if not undetectable: These are "**polymorphic viruses**".

Some antivirus programs use an **integrity checker** to tell if the folders have been changed. The integrity checker builds a database containing information on the executable files on the system (date modified, file size, and possibly a checksum) That way, when an executable file's characteristics change, the antivirus program warns the machine's user. The heuristic method involves analysing the behavior of applications in order to detect actvity similar to that of a known virus. This kind of antivirus program can therefore detect viruses even when the antivirus database has not been updated. On the other hand, they are prone to triggering false alarms.

3. Virus Structure and Operation

True viruses have two major components: one that handles the spread of the virus, and a “payload” or “manipulation” task. The payload task may not be present (has null effect), or it may await a set of predetermined circumstances before triggering. For a computer virus to work, it somehow must add itself to other executable code. The viral code is usually executed before the code of its infected host (if the host code is ever executed again).
One form of classification of computer viruses is based on the three ways a virus may add itself to host code: as a shell, as an add-on, and as intrusive code. A fourth form, the so-called *companion virus*, is not really a virus at all, but a form of *Trojan horse* that uses the execution path mechanism to execute in place of a normal program. Unlike all other viral forms, it does not alter any existing code in any fashion: companion viruses create new executable

files with a name similar to an existing program, and chosen so that they are normally executed prior to the "real" program. As companion viruses are not real viruses unless one uses a more encompassing definition of virus, they will not be described further here.

Shell viruses: A shell virus is one that forms a "shell" (as in "eggshell" rather than "Unix shell") around the original code. In effect, the virus becomes the program, and the original host program becomes an internal subroutine of the viral code. An extreme example of this would be a case where the virus moves the original code to a new location and takes on its identity. When the virus is finished executing, it retrieves the host program code and begins its execution. Almost all boot program viruses (described below) are shell viruses.

Add-on viruses: Most viruses are add-on viruses. They function by appending their code to the host code, and/or by relocating the host code and inserting their own code to the beginning. The add-on virus then alters the startup information of the program, executing the viral code before the code for the main program. The host code is left almost completely untouched; the only visible indication that a virus is present is that the file grows larger, if that can indeed be noticed.

Intrusive viruses: Intrusive viruses operate by overwriting some or all of the original host code with viral code. The replacement might be selective, as in replacing a subroutine with the virus, or inserting a new interrupt vector and routine. The replacement may also be extensive, as when large portions of the host program are completely replaced by the viral code. In the latter case, the original program can no longer function properly. Few viruses are intrusive viruses.

- ➔ A second form of classification used by some authors is to divide viruses into file infectors and boot (system startup) program infectors. This is not particularly clear, however, as there are viruses that spread by altering system-related code that is neither boot code nor programs. Some viruses target file system directories, for example. Other viruses infect both application files *and* boot sectors. This second form of classification is also highly specific and only makes sense for machines that have infectable (writable) boot code.
- ➔ Yet a third form of classification is related to how viruses are activated and select new targets for alteration. The simplest viruses are those that run when their "host" program is run, select a target program to modify, and then transfer control to the host. These viruses are *transient* or *direct* viruses, known as such because they operate only for a short time, and they go directly to disk to seek out programs to infect.
- ➔ The most "successful" PC viruses to date exploit a variety of techniques to remain resident in memory once their code has been executed and their host program has terminated. This implies that, once a single infected program has been run, the virus potentially can spread to any or all programs in the system. This spreading occurs during the entire work session (until the system is rebooted to clear the virus from memory), rather than during a small period of time when the infected program is executing viral code. These viruses are *resident* or *indirect* viruses, known as such because they stay resident in memory, and indirectly find files to infect as they are referenced by the user. These viruses are also known as **TSR (Terminate and Stay Resident)** viruses. If a virus is present in memory after an application exits, how does it remain active? That is, how does the virus continue to infect other programs? The answer for personal computers running software such as MS-DOS is that the virus alters the standard interrupts used by DOS and the BIOS (Basic Input/Output System). The change to the environment is such that the virus code is invoked by other applications when they make service requests.
- ➔ The PC uses many interrupts (both hardware and software) to deal with asynchronous events and to invoke system functions. All services provided by the BIOS and DOS are invoked by the user storing parameters in machine registers, then causing a software interrupt. When an interrupt is raised, the operating system calls the routine whose address it finds in a special table known as the *vector* or *interrupt* table. Normally, this table contains pointers to handler routines in the ROM or in memory-resident portions of the DOS (see figure 4). A virus can modify this table so that the interrupt causes viral code (resident in memory) to be executed. By trapping the keyboard interrupt, a virus can arrange to intercept the CTRL-ALT-DEL soft reboot command, modify user keystrokes, or be invoked on each keystroke. By trapping the `BIOS` disk interrupt, a virus can intercept all `BIOS` disk activity, including reads of boot sectors, or disguise disk accesses to infect as part of a user's disk request. By trapping the `DOS` service interrupt, a virus can intercept all `DOS` service requests including program execution, `DOS` disk access, and memory allocation requests.

- ➔ A typical virus might trap the DOS service interrupt, causing its code to be executed before calling the real DOS handler to process the request. Once a virus has infected a program or boot record, it seeks to spread itself to other programs, and eventually to other systems. Simple viruses do no more than this, but most viruses are not simple viruses. Common viruses wait for a specific triggering condition, and then perform some activity. The activity can be as simple as printing a message to the user, or as complex as seeking particular data items in a specific file and changing their values. Often, viruses are destructive, removing files or reformatting entire disks. Many viruses are also faulty and may cause unintended damage.
- ➔ The conditions that trigger viruses can be arbitrarily complex. If it is possible to write a program to determine a set of conditions, then those same conditions can be used to trigger a virus. This includes waiting for a specific date or time, determining the presence or absence of a specific set of files (or their contents), examining user keystrokes for a sequence of input, examining display memory for a specific pattern, or checking file attributes for modification and permission information. Viruses also may be triggered based on some random event. One common trigger component is a counter used to determine how many additional programs the virus has succeeded in infecting—the virus does not trigger until it has propagated itself a certain minimum number of times. Of course, the trigger can be any combination of conditions.

Computer viruses can infect any form of writable storage, including hard disk, floppy disk, tape, optical media, or memory. Infections can spread when a computer is booted from an infected disk, or when an infected program is run. This can occur either as the direct result of a user invoking an infected program, or indirectly through the system executing the code as part of the system boot sequence or a background administration task. It is important to realize that often the chain of infection can be complex and convoluted. With the presence of networks, viruses can also spread from machine to machine as executable code containing viruses is shared between machines.

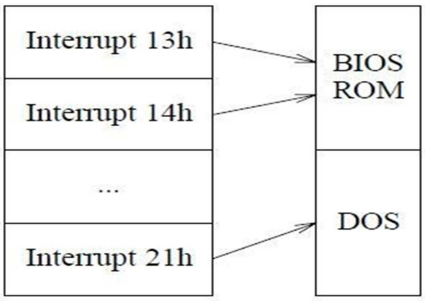

Fig. 1 Normal Interrupt usage

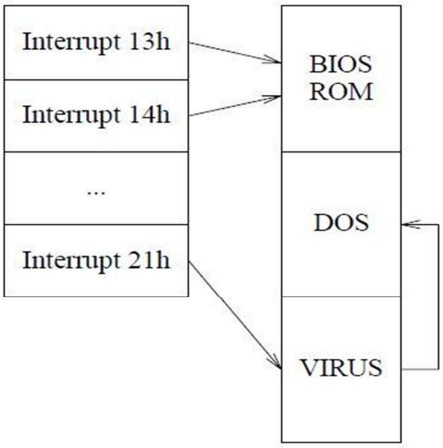

Fig. 2 Interrupt vector with TSR virus

Once activated, a virus may replicate into only one program at a time, it may infect some randomly-chosen set of programs, or it may infect every program on the system. Sometimes a virus will replicate based on some random event or on the current value of the clock. The different methods will not be presented in detail because the result is the same: there are additional copies of the virus on the system.

4. Structure of a TSR

TSRs consist of two distinct parts that execute at different times.

The first part is the installation section, which executes only once, when MS-DOS loads the program. The installation code performs any initialization tasks required by the TSR and then exits through the terminate-and-stay-resident function.

The second part of the TSR, called the resident section, consists of code and data left in memory after termination. Though often identified with the TSR itself, the resident section makes up only part of the entire program.

The TSR's resident code must be able to regain control of the processor and execute after the program has terminated. Methods of executing a TSR are classified as either passive or active.

4.1 Passive TSRs

The simplest way to execute a TSR is to transfer control to it explicitly from another program. Because the TSR in this case does not solicit processor control, it is said to be passive. If the calling program can determine the TSR's memory address, it can grant control via a far jump or call. More commonly, a program activates a passive TSR through a software interrupt. The installation section of the TSR writes the address of its resident code to the proper position in the interrupt vector table ("MS-DOS Interrupts"). Any subsequent program can then execute the TSR by calling the interrupt.

Passive TSRs often replace existing software interrupts. For example, a passive TSR might replace Interrupt 10h, the BIOS video service. By intercepting calls that read or write to the screen, the TSR can access the video buffer directly, increasing display speed.

Passive TSRs allow limited access since they can be invoked only from another program. They have the advantage of executing within the context of the calling program, and thus run no risk of interfering with another process. Such a risk does exist with active TSRs.

4.2 Active TSRs

The second method of executing a TSR involves signaling it through some hardware event, such as a predetermined sequence of keystrokes. This type of TSR is "active" because it must continually search for its startup signal. The advantage of active TSRs lies in their accessibility. They can take control from any running application, execute, and return, all on demand.

An active TSR, however, must not seize processor control blindly. It must contain additional code that determines the proper moment at which to execute. The extra code consists of one or more routines called "interrupt handlers," described in the following section.

5. Interrupt Handlers in Active TSRs

The memory-resident portion of an active TSR consists of two parts. One part contains the body of the TSR - the code and data that perform the program's main tasks. The other part contains the TSR's interrupt handlers.

An interrupt handler is a routine that takes control when a specific interrupt occurs. Although sometimes called an "interrupt service routine," a TSR's handler usually does not service the interrupt. Instead, it passes control to the original interrupt routine, which does the actual interrupt servicing.

Collectively, interrupt handlers ensure that a TSR operates compatibly with the rest of the system. Individually, each handler fulfills one or more of the following functions:

>Auditing hardware events that may signal a request for the TSR

>Monitoring system status

> Determining whether a request for the TSR should be honoured, based on current system status

5.1Auditing Hardware Events for TSR Requests

Active TSRs commonly use a special keystroke sequence or the timer as a request signal. A TSR invoked through one of these channels must be equipped with handlers that audit keyboard or timer events.

A keyboard handler receives control at every keystroke. It examines each key, searching for the proper signal or "hot key." Generally, a keyboard handler should not attempt to call the TSR directly when it detects the hot key. If the TSR cannot safely interrupt the current process at that moment, the keyboard handler is forced to exit to allow the process to continue. Since the handler cannot regain control until the next keystroke, the user has to press the hot key repeatedly until the handler can comply with the request.

Instead, the handler should merely set a request flag when it detects a hot-key signal and then exit normally. Examples in the following paragraphs illustrate this technique.

For computers other than MCA (IBM PS/2 and compatible), an active TSR audits keystrokes through a handler for Interrupt 09, the keyboard interrupt.

Setting a request flag in the keyboard handler allows other code, such as the timer handler (Interrupt 08), to recognize a request for the TSR. The timer handler gains control at every timer interrupt, which occurs an average of 18.2 times per second.

5.2Monitoring System Status

A TSR that uses a hardware device such as the video or disk must not interrupt while the device is active. A TSR monitors a device by handling the device's interrupt. Each interrupt handler simply sets a flag to indicate the device is in use, and then clears the flag when the interrupt finishes.

Only hardware used by the TSR requires monitoring. For example, a TSR that performs disk input/output (I/O) must monitor disk use through Interrupt 13h. The disk handler sets an active flag that prevents the TSR from executing during a read or write operation. Otherwise, the TSR's own I/O would move the disk head. This would cause the suspended disk operation to continue with the head incorrectly positioned when the TSR returned control to the interrupted program.

In the same way, an active TSR that displays to the screen must monitor calls to Interrupt 10h. The Interrupt 10h BIOS routine does not protect critical sections of code that program the video controller. The TSR must therefore ensure it does not interrupt such non rentrant operations.

The activities of the operating system also affect the system status. With few exceptions, MS-DOS functions are not reentrant and must not be interrupted. However, monitoring MS-DOS is somewhat more complicated than monitoring hardware.

5.3Determining Whether to Invoke the TSR

Once a handler receives a request signal for the TSR, it checks the various active flags maintained by the handlers that monitor system status. If any of the flags are set, the handler ignores the request and exits. If the flags are clear, the handler invokes the TSR, usually through a near or far call. How a timer handler detects a request and then periodically scans various active flags until all the flags are clear.

A TSR that changes stacks must not interrupt itself. Otherwise, the second execution would overwrite the stack data belonging to the first. A TSR prevents this by setting its own active flag before executing. A handler must check this flag along with the other active flags when determining whether the TSR can safely execute.

6. Preventing Interference

This section describes how an active TSR can avoid interfering with the process it interrupts. Interference occurs when a TSR commits an error or performs an action that affects the interrupted process after the TSR returns. Examples of interference range from relatively harmless, such as moving the cursor, to serious, such as overrunning a stack.

Although a TSR can interfere with another process in many different ways, protection against interference involves only three steps:

1. Recording a current configuration

2. Changing the configuration so it applies to the TSR

3. Restoring the original configuration before terminating

More sophisticated TSRs may require more sophisticated methods. However, non interference methods generally fall into one of the following categories:

- Trapping errors
- Preserving an existing condition
- Preserving existing data

6.1Trapping Errors

A TSR committing an error that triggers an interrupt must handle the interrupt to trap the error. Otherwise, the existing interrupt routine, which belongs to the underlying process, would attempt to service an error the underlying process did not commit.

For example, a TSR that accepts keyboard input should include handlers for Interrupts 23h and 1Bh to trap keyboard break signals. When MS-DOS detects CTRL+C from the keyboard or input stream, it transfers control to Interrupt 23h (CTRL+C Handler). Similarly, the BIOS keyboard routine calls Interrupt 1Bh (CTRL+BREAK Handler) when it detects a CTRL+BREAK key combination. Both routines normally terminate the current process.

A TSR that calls MS-DOS should also trap critical errors through Interrupt 24h (Critical Error Handler). MS-DOS functions call Interrupt 24h when they encounter certain hardware errors. The TSR must not allow the existing interrupt routine to service the error, since the routine might allow the user to abort service and return control to MS-DOS. This would terminate both the TSR and the underlying process. By handling Interrupt 24h, the TSR retains control if a critical error occurs.

6.2Preserving an Existing Condition

A TSR and its interrupt handlers must preserve register values so that all registers are returned intact to the interrupted process. This is usually done by pushing the registers onto the stack before changing them, then popping the original values before returning.

Setting up a new stack is another important safeguard against interference. A TSR should usually provide its own stack to avoid the possibility of overrunning the current stack. Exceptions to this rule are simple TSRs such as the sample program ALARM that make minimal stack demands.

A TSR that alters the video configuration should return the configuration to its original state upon return. Video configuration includes cursor position, cursor shape, and video mode. The services provided through Interrupt 10h enable a TSR to determine the existing configuration and alter it if necessary.

However, some applications set video parameters by directly programming the video controller. When this happens, BIOS remains unaware of the new configuration and consequently returns inaccurate information to the TSR. Unfortunately, there is no solution to this problem if the controller's data registers provide write-only access and thus cannot be queried directly.

6.3Preserving Existing Data

A TSR requires its own disk transfer area (DTA) if it calls MS-DOS functions that access the DTA. These include file control block functions and Functions 11h, 12h, 4Eh, and 4Fh. The TSR must switch to a new DTA to avoid overwriting the one belonging to the interrupted process. On becoming active, the TSR calls Function 2Fh to obtain the address of the current DTA. The TSR stores the address and then calls Function 1Ah to establish a new DTA. Before returning, the TSR again calls Function 1Ah to restore the address of the original DTA.

MS-DOS versions 3.1 and later allow a TSR to preserve extended error information. This prevents the TSR from destroying the original information if it commits an MS-DOS error. The TSR retrieves the current extended error data by calling MS-DOS Function 59h. It then copies registers AX, BX, CX, DX, SI, DI, DS, and ES to an 11-word data structure in the order given. MS-DOS reserves the last three words of the structure, which should each be set to zero.

7. Conclusion

Our study shows that we should 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 modelling life. For at least this reason, research into computer viruses (using the term in a broader sense, ala Cohen) may be of some scientific interest. By modelling behaviour 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 Live ware may also prove to be of ultimate value. Research into issues on virus defence 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.
To experiment with computer viruses is akin to experimenting with smallpox or anthrax microbes — there may be scientific knowledge to be gained, but the potential for disastrous consequences looms large.

In one sense, we use “computer viruses” every day. Editors, compilers, backup utilities, and other common software meet some definitions of viruses. However, their general nature is known to their users, and they do not operate without at least the implied permission of those users. Furthermore, their replication is generally under the close control or observation of their users. It is these differences from the colloquial computer virus that makes the latter so interesting, however. These differences are also precisely what suggest that computer viruses approach a form of artificial life.

8. Acknowledgements

Much thanks to our guide for his Constructive criticism, and assistance towards the successful completion of this research work**.**

9. References

[1] Leonard Adleman. An abstract theory of computer viruses. In *Lecture Notes in Computer Science, vol 403*. Springer-Verlag, 1990.

[2] David Ferbrache. *A Pathology of Computer Viruses*. Springer-Verlag, 1992.

[3] John Norstad. *Disinfectant On-line Documentation*. Northwestern University, 1.8 edition, June 1990.

[4] Philip Fites, Peter Johnson, and Martin Kratz. *The computer virus crisis*. Van Nostrand Reinhold, 2nd edition, 1992.

[5] Tom Duff. Experiences with viruses on Unix systems. *Computing Systems*, 2(2), Spring 1989.

[6] Frederick B. Cohen. Friendly contagion: Harnessing the subtle power of computer viruses. *The Sciences*, pages 22–28, Sep/Oct 1991

[7] Lance J.Hoffman, editor. *Rogue Programs:Viruses,Worms, and Trojan Horses*. VanNostrand Reinhold, New York, NY, 1990.

[8] Jan Hruska. *Computer Viruses and Anti-VirusWarfare*. Ellis Horwood, Chichester, England, 1990.