



VIEWCORE PROJECT - FALL 2003

Independent Study on Core Dumps

by Atanas Mitkov Dimitrov

Email: atanas_dimitrov@bobcat.gcsu.edu

Project URL: <http://turing.gcsu.edu/~adimitro/viewcore>

TABLE OF CONTENTS:

I. SUMMARY AND TECHNICAL INFORMATION.....	1
1. Objective and summary.....	1
2. Technical information.....	1
II. INTRODUCTION.....	2
1. <i>Programming in C and brief introduction to Linux system calls</i>	2
2. <i>What is a binary file?</i>	2
3. <i>Object file types</i>	2
4. <i>Short Introduction to ELF</i>	3
5. <i>Introduction to gcc and ld</i>	3
6. <i>Introduction to gdb</i>	3
7. <i>Introduction to signals</i>	4
8. <i>Introduction to core dumps</i>	4
III. ELF INTERNALS.....	5
1. <i>ELF header</i>	5
2. <i>ELF Sections</i>	6
3. <i>ELF Segments</i>	7
4. <i>Note Segment</i>	8
5. <i>Brief discussion on assembly and x86 architecture under Linux</i>	8
IV. COREVIEW. CORE DUMP INTERNALS.....	9
1. <i>File information</i>	9
2. <i>Mapping a binary file into main memory</i>	9
3. <i>The ELF header</i>	9
4. <i>Core file segments</i>	10
5. <i>NOTE segment</i>	10
6. <i>PRSTATUS note</i>	10
7. <i>PRPSINFO note</i>	12
8. <i>Other segments and future plans for viewcore's development</i>	12
V. COMPILING WITH GCC AND DEBUGGING CORE DUMPS USING GDB.....	13
1. <i>Compiling with gcc</i>	13
2. <i>Debugging core dumps with gdb</i>	13
APPENDIX A: SAMPLE VIEWCORE OUTPUT.....	15
APPENDIX B: LIST OF SOURCES.....	17
APPENDIX C: VIEWCORE ADDITIONAL INFORMATION.....	17
APPENDIX D: VIEWCORE SOURCE CODE.....	18

I. SUMMARY AND TECHNICAL INFORMATION

1. Objective and summary.

The objective of this paper is to illustrate the various techniques used when debugging core dumps. This paper is a part of an Independent Study class, which I took Fall 2003 at Georgia College and State University, the topic of which was suggested by Dr. Gerald Adkins - my instructor for this class. My classmates in the System Programming class, which I took Spring 2003, were continuously having troubles with software, which they had to write for homework, dumping core. The way Unix handles this process doesn't reveal anything about what could be the reason for this abnormal program termination. The original idea for the Independent Study and the paper was to exemplify the techniques involved when debugging core dumps using `gdb`. However, due to the broad character of concepts and material involved beyond just `gdb`'s perspective of things, the topic was extended to include the core file internals and the methods for acquiring the information stored in such files. `gdb` is an excellent software but the information that it provides about core dumps is only a small fraction of what really hides within a core file, thus one needs software which goes beyond what `gdb` has to offer. This idea inspired `viewcore` - a core dump file debugging software. Currently `viewcore` only works on x86 Linux platforms, which are configured to use the ELF binary format but future plans involve porting it to other operating systems, processor architectures, and binary file formats. The paper begins with a brief introduction of different utilities and formats which were used throughout the study. In Part III I discussed some of the the Executable and Linkable Format (ELF) components which are related to core files. Part IV deals with the core file specific components and the techniques which can be used to acquire the information stored in them. Last, I discuss the methods of debugging core dumps using `gcc` and `gdb`. This paper is not intended to be an all-in-one reference material. Some of the concepts involved are mentioned in a non-lengthy manner and sources for more complete information are specified where I thought one could possibly become eager to learn more about the complete "picture" of how things work and what they do. The various sources are listed in APPENDIX B and the majority should be already present on any machine running current version of Linux.

2. Technical Information.

This section aims at introducing the technical specifications of the equipment and software used at the time of this writing. All experiments, observations, and compiling was performed under Red Hat Linux 8.0 and 9.0 i686, kernel 2.4.20, kernel-source-2.4.20, glibc 2.3.2, gcc 3.2.2. The machine on which the code included in this writing was developed and compiled is Dell Dimension 4400, Pentium IV at 1.7 GHz, 256 MB RAM, single 20GB HD.

I. INTRODUCTION

1. Programming in C and brief introduction to Linux system calls.

The C programming language was developed at Bell Labs during the 1970's. It evolved from a computer language named B and from an earlier language called BCPL. Initially it was designed as a system programming language under UNIX. As the language further developed and standardized, a version known as ANSI (American National Standards Institute) C became dominant. C is still used for most system and network programming as well as for embedded systems. More importantly, there is still a tremendous amount of software still coded in this language and this software is actively maintained. This writing will assume intermediate knowledge of the C programming language. For more information on C's syntax please refer to: <http://www.cs.cf.ac.uk/Dave/C/CE.html>.

System calls are low level system functions, used within a program, which are part of the kernel. System calls are the user's bridge between user space and kernel space. This also means that they are the bridge between a user application and the system hardware. System calls are associated with a unique identifier which tells the kernel what to execute. There is not only one function that handles a particular system call but the execution of a single system call may involve the interleaving of several sets of instructions. Processing a system call is roughly equivalent to handling interrupts and exceptions: general purpose registers are pushed on to the stack, then the control is passed to the system call handler, which then decides which function from the kernel to execute based on the identifier. For more information on system calls and associated identifiers please refer to: `arch/i386/entry.S`

2. What is a binary file?

Binary file is a file which contains information stored in the form of binary strings which are usually meaningless without the appropriate interpreter. In this writing, references to binary files refer to the Executable and Linking Format. All executable files are in binary format. The reason for this is the way that processors execute instructions - in binary format. A common misconception is that a binary file represents ASCII coded instructions. This is NOT the case. Each processor instruction is represented by a unique binary sequence. Each reference to memory and general purpose registers is also in a binary form. For the purpose of this writing addresses will be shortened to hexadecimal (base-16).

3. Object file types.

There are many object file format standards which are implemented by different operating systems vendors. `a.out` is the predecessor to all binary formats and was used in the oldest releases of Unix. Later in SVR3 COFF (Common Object File Format) was used. Today most Linux and Unix flavors migrated to the ELF (Executable and Linkable Format). Darwin (MAC OS X) uses the mach-o runtime architecture. Windows uses Portable Executable (PE) for executable files and COFF for object files.

4. Short Introduction to ELF.

The Executable and Linkable Format (ELF) is the most commonly used object file type among all Unix flavors including Linux. The ELF standard is widely accepted as the default file type in many operating systems because of its power and flexibility. It allows dynamic linking, dynamic loading, runtime control of a process, and improved methodologies for the creation of shared libraries. Additionally the control data is platform independent, which is a great advantage over rest of the formats. The ELF format first started gaining popularity in the early 1990's when different distros began to prompt at installation time whether the user would like to accept the format as the default. Unlike early DOS binaries ELF binaries contain sections in order to structure their contents. I will go in to more detailed description of ELF in Part II.

5. Introduction to gcc and ld.

gcc (GNU C and C++ Compiler) is the most popular compiler used on Unix systems. Upon invocation, gcc performs preprocessing, compilation, assembly, and linking. The process is user controlled and can be interrupted at any stage by passing compiler arguments. Each stage can further be controlled to allow even greater flexibility to software developers. In this writing one option is of special significance. When a programmer includes the `-g` option then gcc includes various information within the object file for debugging purposes. This information is in the system's native format: stabs, COFF, XCOFF, or DWARF. Further, the programmer can force a specific type of debugging information format to be used, among which gdb (which I will discuss in the next section) specific format. The usefulness of this option in regards to core dumps will be discussed in later sections.

The final process of using gcc against a source file is linking. This is usually done by ld. ld combines a number of object and archive files, relocates their data and ties up symbol references. ld uses AT&T's Link Editor Command Language syntax instructions to provide total control over the linking process. Also, upon linking, ld provides diagnostic information. Although it exists as an autonomous executable ld is mostly invoked indirectly as the final stage of compiling. In this case it can still be passed options by using `-wl <option>` format. For more information on gcc and ld please refer to their manual pages.

6. Introduction to gdb.

GDB (GNU Debugger) is the most commonly used debugger on Unix platforms. It makes possible for the developer to see what is actually happening within the running process or to display information about what caused it to terminate abnormally. The first of the functions is performed largely by the use of functions similar to `ptrace`, which allows for a process to attach to a running process and view and change any of the information (stored in binary form in memory) of its core image. This technique is also used in patching running processes (in-core patching). The next feature of gdb which will be of greater interest of this writing, is the information which gdb can supply in regards to the abnormal termination of a process. In cases as such, the information generated by gcc by using the `-g` flag can be very helpful combined with the use of gdb. The debugger is then able to pinpoint with great precision the reason for the

sudden death of a process. gdb 's functionality in regards to core dumps will be discussed in more detail in Part V. For more general information on gdb please refer to its manual page.

7. Introduction to signals.

One way of inter-process communication is the use of signals. Signals allow for a number of actions to be taken upon their receipt. Here is a listing of the original POSIX.1 signal definitions:

SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGSEGV	11	Core	Invalid memory reference

Core dumps are created by the kernel upon the receipt of a certain type of signal by a process before it is removed from the processor queue and main memory altogether. Such signals are indicated by `Core` in the POSIX.1 excerpt. After 2001 revisions of the standard more signal types were added, thus increasing the number of signals upon which the core image of the process is dumped on to secondary memory:

SIGBUS	10,7,10	Core	Bus error (bad memory access)
SIGSYS	12,-,12	Core	Bad argument to routine (SVID)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2 BSD)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2 BSD)
SIGIOT	6	Core	IOT trap. A synonym for SIGABRT
SIGEMT		Core	

Understanding of signals is important when debugging processes. For more complete information on signals visit the manual page for `signal(7)`.

8. Introduction to core dumps.

Core dumps are special types of binary files which are created upon the abnormal termination of a process currently being executed. They contain a significant amount of information such that when parsed to a debugger together with the executable that caused it (especially when the executable is compiled with the debugging option) enables the developer to determine the cause for error that "terminated" the process. Core dumps will be discussed in greater detail in Part IV.

III. ELF INTERNALS

Most of the material that I will cover about core dumps will be meaningless without discussion of how executables are kept on disk under Linux. Core dumps are created in a manner that complies with the ELF standard.

There are mainly three different types of ELF files:

- relocatable files, which hold information used for linking multiple object files in order to create shared object files or executables.
- shared object files, which contain information used in the two different possible views of an ELF file. These files cannot be executed but just combined with executables.
- and executable files, which are files in format suitable for execution. They tell `exec()` how to create a process image in memory.

The two different views for ELF files are linking and loading (execution) view. The linking view is static, very meaningful view and is designed for the linker to completely describe the ELF object file internals. The loading view is much simpler and is used at runtime to load executables and shared objects and make up a running process.

1. ELF header

The first part of any ELF object file is the ELF header. The definition of an ELF header is normally included in `include/elf..` Here is how an ELF header looks like under Linux:

```
typedef struct
{
    unsigned char e_ident[EI_NIDENT];    /* Magic number and other info */
    Elf32_Half    e_type;                 /* Object file type */
    Elf32_Half    e_machine;              /* Architecture */
    Elf32_Word    e_version;              /* Object file version */
    Elf32_Addr    e_entry;                 /* Entry point virtual address */
    Elf32_Off     e_phoff;                 /* Program header table file offset*/
    Elf32_Off     e_shoff;                 /* Section header table file offset*/
    Elf32_Word    e_flags;                 /* Processor-specific flags */
    Elf32_Half    e_ehsize;                /* ELF header size in bytes */
    Elf32_Half    e_phentsize;            /* Program header table entry size */
    Elf32_Half    e_phnum;                /* Program header table entry count*/
    Elf32_Half    e_shentsize;            /* Section header table entry size */
    Elf32_Half    e_shnum;                /* Section header table entry count*/
    Elf32_Half    e_shstrndx;             /* Section header string table index*/
} Elf32_Ehdr;
```

* Half stands for 2 bytes, Addr is 4 bytes, Off (offset address) 4 bytes, Word is 4 bytes.

* 32 describes the particular processor (in this case 32-bit, definition also exists for 64-bit)

`e_ident` contains not only the magic number but also the processor class (64 or 32 bit), and the data encoding type which is also processor specific. Possible values for data encoding

types are ELFDATANONE (0), ELFDATA2LSB (1), ELFDATA2MSB (2). LSB specifies that the least significant bit occupies the lowest address and MSB specifies that the most significant bit is located at the lowest address. For example Intel 32-bit architecture will require ELFCLASS32 and ELFDATA2LSB.

e_type can be any of the following:

```
#define ET_NONE          0          /* No file type */
#define ET_REL          1          /* Relocatable file */
#define ET_EXEC         2          /* Executable file */
#define ET_DYN          3          /* Shared object file */
#define ET_CORE         4          /* Core file */
#define ET_NUM          5          /* Number of defined types */
#define ET_LOOS         0xfe00     /* OS-specific range start */
#define ET_HIOS         0xfeff     /* OS-specific range end */
#define ET_LOPROC      0xff00     /* Processor-specific range start */
#define ET_HIPROC      0xffff     /* Processor-specific range end */
```

It is clear that the ELF header contains an enormous amount of useful information which is used by the two views. To have a better understanding of what exactly is stored in an ELF file I will discuss the various different "sections".

2. ELF Sections.

Sections are mostly used in the linking view. The section header table contains array of section headers each of which has the following form:

```
typedef struct
{
    Elf32_Word    sh_name;          /* Section name (string tbl index) */
    Elf32_Word    sh_type;          /* Section type */
    Elf32_Word    sh_flags;        /* Section flags */
    Elf32_Addr    sh_addr;          /* Section virtual addr at execution */
    Elf32_Off     sh_offset;        /* Section file offset */
    Elf32_Word    sh_size;          /* Section size in bytes */
    Elf32_Word    sh_link;          /* Link to another section */
    Elf32_Word    sh_info;          /* Additional section information */
    Elf32_Word    sh_addralign;     /* Section alignment */
    Elf32_Word    sh_entsize;      /* Entry size if section holds table */
} Elf32_Shdr;
```

Here are some of the most common section types (sh_type):

```
#define SHT_NULL        0          /* Section header table entry unused */
#define SHT_PROGBITS    1          /* Program data */
#define SHT_SYMTAB      2          /* Symbol table */
#define SHT_STRTAB      3          /* String table */
#define SHT_RELA        4          /* Relocation entries with addends */
```

```

#define SHT_HASH          5          /* Symbol hash table */
#define SHT_DYNAMIC      6          /* Dynamic linking information */
#define SHT_NOTE         7          /* Notes */
#define SHT_NOBITS       8          /* Program space with no data (bss)*/
#define SHT_REL          9          /* Relocation entries, no addends */
#define SHT_SHLIB        10         /* Reserved */
#define SHT_DYNSYM       11         /* Dynamic linker symbol table */
#define SHT_INIT_ARRAY   14         /* Array of constructors */
#define SHT_FINI_ARRAY   15         /* Array of destructors */

```

All the section information is meaningful at the static loading view. The sections which are not needed later are stripped off after all their information has been distributed to the appropriate handlers. For more complete information on various sections please refer to PFS (Portable Formats Specification) of the ELF by TIS (Tool Interface Standards).

3. ELF Segments

Elf segments are described by some of the program headers. There is a program header for each segment. Segments are meaningful to the executable view. Here is an example how a program header looks like under Linux:

```

typedef struct
{
    Elf32_Word    p_type;          /* Segment type */
    Elf32_Off     p_offset;        /* Segment file offset */
    Elf32_Addr    p_vaddr;        /* Segment virtual address */
    Elf32_Addr    p_paddr;        /* Segment physical address */
    Elf32_Word    p_filesz;       /* Segment size in file */
    Elf32_Word    p_memsz;        /* Segment size in memory */
    Elf32_Word    p_flags;        /* Segment flags */
    Elf32_Word    p_align;        /* Segment alignment */
} Elf32_Phdr;

```

Some of the most common types of program segments are:

```

#define PT_NULL        0          /* Program header table entry unused */
#define PT_LOAD        1          /* Loadable program segment */
#define PT_DYNAMIC     2          /* Dynamic linking information */
#define PT_INTERP      3          /* Program interpreter */
#define PT_NOTE        4          /* Auxiliary information */
#define PT_SHLIB       5          /* Reserved */
#define PT_PHDR        6          /* Entry for header table itself */

```

* PT_NOTE is of special interest in regards to core dumps.

For more complete information on various segments please refer to PFS (Portable Formats Specification) of the ELF by TIS (Tool Interface Standards).

4. Note Segment.

Note sections contain rather interesting amount of information in regards to core dumps. On occasions certain vendor may or may not wish to include this type of sections (type `SHT_NOTE`) within the binary. The presence of this type of sections is optional and affects neither the linking nor loading or relocating. It is generally used for compatibility and conformance checks, but in our case this will be excellent means for storing useful information. The note section is an array of 4 byte words. There are five distinguishable members within a note section: `namesz`, `descsz`, `type`, `name`, and `desc`. Before going into more details here is how the "note" looks like in an object file (or core dump) (Fig. X).

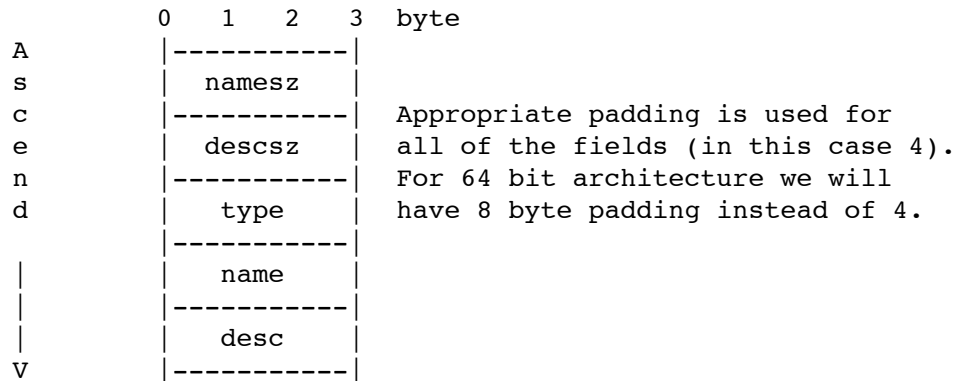


Fig. X

The note's name contains the name of the note owner, which could be any string. The description is the field that core dump analysis can profit from. This field can contain all sorts of information and the contents of this field may vary with different OS vendors and processor architectures. Members `namesz`, and `descsz` are used to store the limits of the information stored in `name` and `desc` respectively. The `type` field only further specifies the nature of the information stored in a note. Here is how the note section looks like on Linux under `include/elf.h`:

```
typedef struct
{
    Elf32_Word n_namesz;           /* Length of the note's name. */
    Elf32_Word n_descsz;         /* Length of the note's descriptor.*/
    Elf32_Word n_type;           /* Type of the note. */
} Elf32_Nhdr;
```

I will discuss in more detail the contents of this type of section in regards to core dumps in Part IV.

5. Brief discussion on assembly and x86 architecture under Linux.

Before I continue with the core dump internals I would like to address some of the notation that will be used in the following sections of this writing. In GAS (The GNU Assembler) the registers are addressed as `%reg`, where `reg` is the name of the register and the percentage sign is the standard way to indicate such entity. `e` in front of register type (such as `%eax`) indicates 32 bits. The program for core dump debugging is written for 32-bit Intel architecture which enforces specific amount of padding where applicable and very specific note section information as I will be

discussing later. For more information on the Intel instruction set and registers please refer to: <http://developer.intel.com/design/pentium4/manuals/245470.htm>. For more information on x86 Assembler under Linux please refer the gas manual page or: <http://linuxassembly.org>.

IV. COREVIEW. CORE DUMP INTERNALS.

1. File information.

The first questions that come to mind when examining a particular file on a Unix system are: who is the file owned by, what is its size, when was it modified, and are there any links to it. Thus I will first go over the way of obtaining such information from a core dump. The easiest way in which one can do this is by using the `ls` utility. However, there are alternate ways of obtaining this information. One such method involves the use of `fstat(2)` which is commonly used to retrieve file status information. `fstat(2)` returns the answers to all of the questions above in a single structure, which can be retrieved within a program and whose members can then be the displayed in a readable form.

2. Mapping a binary file into main memory.

In order to retrieve any information from the core dump using a program we must first transfer its contents from secondary memory to main memory. One way of doing this is by using the `mmap(2)` function. `mmap(2)` maps files or devices into memory. After mapping it returns a pointer to the memory region within the process which contains the mapped area. `Viewcore` uses `mmap(2)` to store the file's contents into a single array. This is a common technique used by software such as `strings(1)`. If one were to display the contents of the array to `stdout` then one would be able to see all the contents of the core file interpreted as printable ASCII characters some of which are readable.

3. The ELF header.

The first part of a core dump is its ELF header. The ELF header is in the format of any other object file but it contains information which is specific to core dumps. For example the value for the object file type is 4 (`ET_CORE`), which identifies the binary file as a core file. The ELF header of a core file can be retrieved by simply creating a pointer of type `Elf32_Ehdr` to the starting address of the array. Core files are different than object files in that they are not created with the intention to be executed. Thus the sections headers together with all the sections and relocation information is now not present. Those members of an ELF object file were stripped off when the process was executed. Thus as was discussed earlier we can consider the core file to be more similar to the executable view of ELF. This seems consistent with the fact that a core dump is a snapshot of the core image of a process which is already running in memory. Since this is the case we have several segments which contain different information which will be useful in demystifying core dumps. Other fields which are present within the core file's ELF header are the processor architecture information, the Application Binary Interface of the operating system, and the object file version.

4. Core file segments.

As can be observed from the sample output in Appendix A, and as was mentioned in the previous section, core dumps contain a number of segments which were written out to secondary memory. We can separate those segments in two groups: core file descriptors and process core image segments. One can examine the contents of these segments in order to gather more information about the process that generated the core file. The next three sections deal with identifying the type of information stored in the segments known at the time of this writing and ways to display it in human readable form.

5. NOTE segment.

The note segment is dumped on to secondary memory as the first ELF segment in a core file. This segment contains information which is quite easy to get and interpret. The type of notes which contain information about the core dump (which debuggers take advantage of) are named CORE. The NOTE segment contains an array of notes (as discussed earlier), which instead of vendor specific data contain information about what exactly was the status of the process and the processor at the time the core was dumped. Those notes are 4 bytes padded (since 32-bit x86 architecture). The number of notes within the NOTE segment is variable based on the OS and processor architecture. Note types vary based on the information enclosed and based on the specific processor architecture and certainly operating system. For example `NT_TASKSTRUCT`, `NT_LWPSTATUS`, `NT_LWPSINFO`, `NT_FPREGSET` are note types which can only be interpreted in a meaningful manner on Solaris. They all contain pointers to structures only defined for the Sun Microsystems OS and particular to the Sparc architecture. There are only two types of notes that are meaningful at the time of this writing and they are discussed in the next two sections.

6. PRSTATUS note.

It is easy to determine by simply looking at its name, that this type of note carries process status information. This information is compiled in a `prstatus` structure which is defined in `include/linux/elfcore.h` as follows:

```
struct elf_prstatus
{
#ifdef 0
    long    pr_flags;          /* XXX Process flags */
    short   pr_why;           /* XXX Reason for process halt */
    short   pr_what;         /* XXX More detailed reason */
#endif
    struct elf_siginfo pr_info; /* Info associated with signal */
    short   pr_cursig;       /* Current signal */
    unsigned long pr_sigpend; /* Set of pending signals */
    unsigned long pr_sighold; /* Set of held signals */
#ifdef 0
    struct sigaltstack pr_altstack; /* Alternate stack info */
    struct sigaction pr_action; /* Signal action for current sig */
#endif
}
```

```

    pid_t   pr_pid;
    pid_t   pr_ppid;
    pid_t   pr_pgrp;
    pid_t   pr_sid;
    struct timeval pr_utime;      /* User time */
    struct timeval pr_stime;      /* System time */
    struct timeval pr_cutime;     /* Cumulative user time */
    struct timeval pr_cstime;     /* Cumulative system time */
#ifdef 0
    long    pr_instr;            /* Current instruction */
#endif
    elf_gregset_t pr_reg;        /* GP registers */
    int pr_fpvalid;             /* True if math co-processor being used. */
};
*Fields which are just present but not used are marked XXX.

```

There are several fields (members of the structure) that are of special interest. First the signal which the process received and caused it to dump core. This signal is stored in `pr_cursig`. Yet another useful information is the `pid`, `ppid`, `pgrp`, and `sid` of the process that caused the core dump. The time in seconds that the process has spent running is also noted using `timeval` structure which is defined in `sys/time.h`.

The last set of information within this type of note is a snapshot of the processor registers at the time of the core dump. They are contained in a structure of type `elf_gregset_t`. In `viewcore` I have defined an alternative equivalent structure to hold those values called `user_regs_struct`:

```

typedef struct
{
    long int ebx;
    long int ecx;
    long int edx;
    long int esi;
    long int edi;
    long int ebp;
    long int eax;
    long int xds;    /* Each member represents a general purpose register */
    long int xes;
    long int xfs;
    long int xgs;
    long int orig_eax;
    long int eip;
    long int xcs;
    long int eflags;
    long int esp;
    long int xss;
}user_regs_struct;

```

7. *PRPSINFO* note.

This type of note displays only general process information. Some of the information such as `pid`, `ppid`, `pgrp`, and `sid` is identical to the information provided by a `PRSTATUS` note. The information of a `PRPSINFO` note is defined in a structure of type `elf_prpsinfo` in `include/linux/elfcore.h` as follows:

```
struct elf_prpsinfo
{
    char    pr_state;        /* numeric process state */
    char    pr_sname;        /* char for pr_state */
    char    pr_zomb;         /* zombie */
    char    pr_nice;         /* nice val */
    unsigned long pr_flag;   /* flags */
    __kernel_uid_t pr_uid;
    __kernel_gid_t pr_gid;
    pid_t   pr_pid, pr_ppid, pr_pgrp, pr_sid;
    /* Lots missing */
    char    pr_fname[16];    /* filename of executable */
    char    pr_psargs[ELF_PRARGSZ]; /* initial part of arg list */
};
```

Note that this time the process' effective user ID and group ID are included. This can pretty much tell us who executed the program. In addition to this the name of the executable that caused the core dump is included.

8. *Other segments and future plans for viewcore's development.*

Note segments are not the only ones present within a core file. There are numerous sections of type `LOAD` which can be seen in the sample output in Appendix A. These segments contain the rest of the process' core image. At the time of this writing only the first one has been identified as the ELF header of the original object file. Immediately follows is the `.text` section dump which contains the set of assembly instructions which the process was supposed to execute. The part of `viewcore` that deals with translating binary into assembly is currently under development. This paper will serve as the information supplied in `doc` directory of the source. `Viewcore` is certainly not the only program out there which is able to do the things it does. Similar projects deal with the ELF object files in general. Such projects are `elfutils` and `binutils`. `Elfutils` is a relatively new set of utilities dedicated to ELF internals written by Red Hat staff. It provides shared libraries which contain just about any function which could be useful in dealing with ELF object files. Similarly `binutils` has almost identical functionality, but certainly different implementation and has been present on Unix and Linux systems for a long time. Regardless of the already existing projects `viewcore` differs radically from any of them due to its main focus - core dumps. There is not any software that I am aware of which dives at such depth into the core file internals. In terms of primary concentration, `viewcore` is unique. Following releases of `viewcore` will aim at porting it to Solaris and Mac OS X. Future plans of `viewcore's` development also include the idea of debugging running processes by attaching to them and dumping their core image on to disk.

V. COMPILING WITH GCC AND DEBUGGING CORE DUMPS USING GDB.

1. *Compiling with gcc.*

As discussed earlier `gcc` is the most commonly used compiler on Unix systems. Developers can profit from using `gcc` because it inserts debugging information which can be used by `gdb` in order to troubleshoot software which dumps core. If a programmer should want to do this the `gcc` must be invoked with `-g` option and any of its accepted values. Thus the command line:

```
$ gcc -g source_file.c
```

will enable this feature. Using any of the accepted values, for example:

```
$ gcc -ggdb source_file.c
```

,will further force the type of debugging information formatting desired.

Sometimes core dumps are predictable simply looking at minor warnings generated by `gcc`, which are not displayed by default. In order to enable `gcc` to output them to `stdout` we use the following:

```
$ gcc -Wall source_file.c
```

Thus one can use the combination:

```
$ gcc -g -Wall source_file.c
```

2. *Debugging core dumps with gdb.*

As discussed earlier `gdb` is the most commonly used debugger on Unix systems. `gdb` is a very powerful software a lot of functionality and most of its features not related to core dumps will not be discussed. For the following example I will use a simple source file written in C which is guaranteed to dump core:

```
int main(void)
{
    int a=9999, b=0;

    int c=a/b;
    return(0);
}
```

We compile the file using:

```
$ gcc -g -Wall source_file.c
```

Now, we want to make sure that core will be dumped. Sometimes systems disable this feature but you can easily change it. If you are using `bash` you can do this by:

```
$ ulimit -c 100
```

or any numeric value which will be the upper bound for the size of your core dump.

Now, we must execute the resulting executable:

```
$ ./a.out
Floating point exception (core dumped)
$
```

As you see the core was dumped!

In order to invoke gdb:

```
$ gdb
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

This is the default prompt for the gdb's interactive shell.

Now you must specify which executable file will be used for acquiring the symbol table:

```
(gdb) file a.out
Reading symbols from a.out...done.
(gdb)
```

Now you must specify the core dump file name

```
(gdb) core-file core.1993
Core was generated by `./a.out'.
Program terminated with signal 8, Arithmetic exception.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x08048318 in main () at test.c:5
5          int c=a/b;
```

And quit the debugger:

```
(gdb) quit
$
```

So the core dump was caused by the instruction at line 5 in the source file where there is an obvious division by 0.

APPENDIX A: SAMPLE VIEWCORE OUTPUT

.: FILE STATUS INFORMATION :.

hard links: 1
uid: 500
gid: 500
size: 49152
block size: 4096
block count: 88
last access: Mon Sep 1 19:36:05 2003
last mod: Fri Aug 22 17:04:02 2003
last change: Fri Aug 22 17:04:02 2003

.: CORE ELF HEADER INFORMATION :.

capacity: 32 bit (ELFCLASS32) [1]
data encoding type: 2's comp, little endian (ELFDATA2LSB) [1]
file version: current (EV_CURRENT) [1]
OS ABI: Unix System V ABI [0]
ABI version: 0
padding bytes: 0
object file type: core file (ET_CORE) [4]
architecture: Intel 80386 (EM_NONE) [3]
object file version: current (EV_CURRENT) [1]
entry point virtual address: 0
program header table file offset: 0x34
section header table file offset: 0
processor specific flags: 0
ELF header size: 52 bytes
program header table entry size: 32 bytes
program header table entry count: 10
section header table entry size: 0 bytes
section header table entry count: 0
section header string table index: 0

.: SEGMENT INFORMATION :.

type	offset	vaddress	paddress	size(file)	size(mem)	RWX	align
NOTE	0x174	0	0	2028	0	0	0
LOAD	0x1000	0x8048000	0	0	4096	0x5	0x1000
LOAD	0x1000	0x8049000	0	4096	4096	0x6	0x1000
LOAD	0x2000	0x40000000	0	0	86016	0x5	0x1000
LOAD	0x2000	0x40015000	0	4096	4096	0x6	0x1000
LOAD	0x3000	0x40016000	0	4096	4096	0x6	0x1000
LOAD	0x4000	0x42000000	0	0	1236992	0x5	0x1000
LOAD	0x4000	0x4212e000	0	12288	12288	0x6	0x1000
LOAD	0x7000	0x42131000	0	8192	8192	0x6	0x1000
LOAD	0x9000	0xbfffd000	0	12288	12288	0x7	0x1000

.: NOTES INFORMATION :.

found note section at offset: 0x174

```
--- note 0 at offset 0x174 ---
padding:                4 bytes
note name size:        0x5 bytes
note description size: 0x90 bytes
note name:             CORE
note type:             PRSTATUS [1]
signal number:        8
extra code:           0
errno:                0
current signal:       8
set of pending signals: 0
set of held signals:  0
pid:                  4843
ppid:                 4843
pgrp:                 4843
sid:                  4813
user time:            0.0 sec
system time:          0.0 sec
cumulative user time: 0.0 sec
cumulative system time: 0.0 sec
bool pr_fpvalid:      0
```

general purpose registers:

```
ebx: 0x42130a14      ecx: 0x42015554      edx: 0
esi: 0x40015360      edi: 0x8048358      ebp: 0xbffffe268
eax: 0x270f          xds: 0x2b          xes: 0x2b
xfs: 0              xgs: 0x33          orig_eax: 0xffffffff
eip: 0x8048318      xcs: 0x23          eflags: 0x10286
esp: 0xbffffe250      xss: 0x2b
```

```
--- note 1 at offset 0x218 ---
padding:                4 bytes
note name size:        0x5 bytes
note description size: 0x7c bytes
note name:             CORE
note type:             PRPSINFO [3]
numeric process state:
char for pr_state:    R
zombie:
nice val:
flag 1:               0
uid:                  500
gid:                  500
pid:                  4843
ppid:                 4813
pgrp:                 4843
sid:                  4813
filename or executable: a.out
```

```
--- note 2 at offset 0x2a8 ---
padding:                4 bytes
note name size:        0x5 bytes
```

note description size: 0x6b0 bytes
note name: CORE
note type: PRXREG [4]
next note at 0x7f8

.: ENCLOSED ELF HEADER INFORMATION at 0x1000 :.
capacity: 32 bit (ELFCLASS32) [1]
data encoding type: 2's comp, little endian (ELFDATA2LSB) [1]
file version: current (EV_CURRENT) [1]
OS ABI: Unix System V ABI [0]
ABI version: 0
padding bytes: 0
object file type: executable file (ET_EXEC) [2]
architecture: Intel 80386 (EM_NONE) [3]
object file version: current (EV_CURRENT) [1]
entry point virtual address: 0x8048244
program header table file offset: 0x34
section header table file offset: 0x1cdc
processor specific flags: 0
ELF header size: 52 bytes
program header table entry size: 32 bytes
program header table entry count: 6
section header table entry size: 40 bytes
section header table entry count: 34
section header string table index: 31

APPENDIX B: LIST OF SOURCES

Websites:

<http://www.cs.cf.ac.uk/Dave/C/CE.html>
<http://developer.intel.com/design/pentium4/manuals/245470.htm>.
<http://linuxassembly.org>.

Linux Source Files:

*arch/i386/entry.S
**include/elf.h
**include/linux/elfcore.h
**sys/time.h
*located in kernel sources root **located in standard sources root

Manual Pages:

gcc(1), ld(1), gdb(1), signal(7), ls(1), fstat(2), mmap(2), strings(1)

Other Documentation:

PFS (Portable Formats Specification) of the ELF by TIS (Tool Interface Standards)

APPENDIX C: VIEWCORE ADDITIONAL INFORMATION

Current Maintainer: Atanas Mitkov Dimitrov
E-mail: atanas_dimitrov@bobcat.gcsu.edu
Project URL: <http://turing.gcsu.edu/~adimitro/viewcore>

APPENDIX D: VIEWCORE SOURCE CODE

```
/*
*****
main.c - core dump debugger
-----
begin          : Sun Aug  1 20:46:02 EDT 2003
copyright     : (C) 2003 by Atanas Dimitrov
email        : atanas_dimitrov@bobcat.gcsu.edu
*****
/

/*
*****
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation; either version 2 of the License, or
* (at your option) any later version.
*
*
*****
/

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <elf.h>
#include <sys/procfs.h>
#include <sys/time.h>
#include <sys/user.h>

#define TRUE 1
#define FALSE 0
#define EMPTY 0
#define MAX_NOTES 10

/* to avoid implicit declaration warnings generated by -Wall */
extern char *ctime(const time_t *timep);

typedef struct
{
    long int ebx;
    long int ecx;
    long int edx;
    long int esi;
    long int edi;
    long int ebp;
    long int eax;
    long int xds;
    long int xes;
    long int xfs;
    long int xgs;
    long int orig_eax;
    long int eip;
    long int xcs;
    long int eflags;
    long int esp;
    long int xss;
}user_regs_struct;

/* need to replace this with include/elf.h: Elf32_Nhdr ... wish I saw it earlier*/
/* this does work out pretty good for the 32 bit arch and accounts for the Note structure. */
/* definition is next to processing section */
typedef struct {
    Elf32_Word namesz;
    Elf32_Word descsz;
    Elf32_Word type;
}Elf32_Note_Part;

int main(int argc, char *argv[])
{
    int fdin;
    char *acctime, *modtime, *ctime;
```

```

char *longarray;
struct stat statbuf;
int os_abi_unknown;
Elf32_Ehdr *elfhdr, *elfhdr1;
Elf32_Phdr *proghdr;
int noteoffsets[MAX_NOTES];
int chk_phoff=0, chk_phentsize=0, chk_phnum=0;
int count, next_off=0;
int note_sect_counter=0; /* number of note sections */
int prstatus_bool=0, prpsinfo_bool=0;

/* check for necessary number of command line arguments */
if(argc !=2)
{
    printf("Usage: %s <binary_file>\n", argv[0]);
    exit(1);
}

/* check for successful binary file opening */
if ((fdin = open(argv[1], O_RDONLY)) == -1)
{
    perror("open");
    exit(2);
}

/* get the attributes of the file but first check for errors */
if ((fstat(fdin, &statbuf) < 0)
{
    perror("fstat");
    exit(3);
}

/* display some attributes */
printf("\n.: FILE STATUS INFORMATION :.\n");
printf("hard links:    %d\n", statbuf.st_nlink);
printf("uid:            %d\n", statbuf.st_uid);
printf("gid:            %d\n", statbuf.st_gid);
printf("size:           %d\n", statbuf.st_size);
printf("block size:     %d\n", statbuf.st_blksize);
printf("block count:    %d\n", statbuf.st_blocks);
acctime = (char *) ctime(&statbuf.st_atime);
printf("last access:   %s", acctime);
modtime = (char *) ctime(&statbuf.st_mtime);
printf("last mod:      %s", modtime);
chtime = (char *) ctime(&statbuf.st_ctime);
printf("last change:  %s", chtime);

//printf("\nstarting to modify output...\n");
fflush(stdout);
//sleep(1);

longarray = mmap(0, statbuf.st_size, PROT_READ, \
    MAP_FILE | MAP_SHARED, fdin, 0);
//perror("\nstatus of memory mapping");

/* use memcmp to compare the first 4 bytes from file to
   first 4 bytes of standard ELF header's first entry
   - Elf32_Ehdr.e_ident[EI_IDENT] basically do byte
   comparison at already defined indexes and/or offsets.
   (see include/elf.h for definitions)
   all this to verify the ELF type
*/

if (memcmp(longarray, ELFMAG, SELFMAG) == 0)
{
    //printf("ELF file type accepted\n");
    fflush(stdout);
}
else
{
    printf("error: unrecognized file type\n");
    fflush(stdout);
    exit(5);
}

/* initialize the elf header structure as defined in include/elf.h */
elfhdr = (Elf32_Ehdr *) ((unsigned char *) &longarray[0x0]);
printf("\n.: CORE ELF HEADER INFORMATION :.\n");
/* determine file class */

```



```

        printf("OS ABI: unknown                [%d]\n", elfhdr->e_ident[EI_OSABI]);
        os_abi_unknown = TRUE;
    }

    /* determine ABI version if applicable */
    if (!os_abi_unknown)
        printf("ABI version:                %d\n", elfhdr->e_ident[EI_ABIVERSION]);
    else
        printf("ABI version:                unknown    [%d]\n", elfhdr-
>e_ident[EI_ABIVERSION]);

    /* display number of padding bytes */
    printf("padding bytes:                %d\n", elfhdr->e_ident[EI_PAD]);

    /* determine object file type */
    if (elfhdr->e_type == ET_NONE)
        printf("object file type:                no file type    (ET_NONE) [%d]\n", \
            elfhdr->e_type);
    else if (elfhdr->e_type == ET_REL)
        printf("object file type:                relocatable file (ET_REL) [%d]\n", \
            elfhdr->e_type);
    else if (elfhdr->e_type == ET_EXEC)
        printf("object file type:                executable file (ET_EXEC) [%d]\n", \
            elfhdr->e_type);
    else if (elfhdr->e_type == ET_DYN)
        printf("object file type:                shared object file (ET_DYN) [%d]\n", \
            elfhdr->e_type);
    else if (elfhdr->e_type == ET_CORE)
        printf("object file type:                core file (ET_CORE) [%d]\n", \
            elfhdr->e_type);
    else if (elfhdr->e_type == ET_NUM)
        printf("object file type:                number of defined types (ET_NUM) [%d]\n", \
            elfhdr->e_type);
    else if (elfhdr->e_type == ET_LOOS)
        printf("object file type:                OS-specific range start (ET_LOOS) [%4x]\n", \
            elfhdr->e_type);
    else if (elfhdr->e_type == ET_HIOS)
        printf("object file type:                OS-specific range end (ET_HIOS) [%4x]\n", \
            elfhdr->e_type);
    else if (elfhdr->e_type == ET_LOPROC)
        printf("object file type:                Processor-specific range start (ET_LOPROC) [%4x]\n", \
            elfhdr->e_type);
    else if (elfhdr->e_type == ET_HIPROC)
        printf("object file type:                Processor-specific range end (ET_HIPROC) [%4x]\n", \
            elfhdr->e_type);
    else
        printf("object file type:                undefined [%4x]\n", elfhdr->e_type);

    /* determine architecture */
    /* 100 definitions enter here... just 3 for right now*/

    if (elfhdr->e_machine == EM_NONE)
        printf("architecture:                none    (EM_NONE) [%d]\n", \
            elfhdr->e_machine);
    else if (elfhdr->e_machine == EM_386)
        printf("architecture:                Intel 80386 (EM_386) [%d]\n", \
            elfhdr->e_machine);
    else if (elfhdr->e_machine == EM_SPARC)
        printf("architecture:                Sun SPARC (EM_SPARC) [%d]\n", \
            elfhdr->e_machine);
    else
        printf("architecture:                unknown [%d]\n", elfhdr->e_machine);

    /* determine object file version */
    if (elfhdr->e_version == EV_NONE)
        printf("object file version:            invalid ELF version (EV_NONE) [%d]\n", \
            elfhdr->e_version);
    else if (elfhdr->e_version == EV_CURRENT)
        printf("object file version:            current (EV_CURRENT) [%d]\n", \
            elfhdr->e_version);
    else printf("object file version:            unknown [%d]\n", elfhdr->e_version);

    /* display the entry point virtual address */
    printf("entry point virtual address:    %#x\n", elfhdr->e_entry);

    /* display the program header table file offset */
    printf("program header table file offset:  %#x\n", elfhdr->e_phoff);
    if (elfhdr->e_phoff > 0)
        chk_phoff=1;

```

```

/* display section header table file offset */
printf("section header table file offset:  %#x\n", elfhdr->e_shoff);

/* display processor-specific flags */
printf("processor specific flags:          %#x\n", elfhdr->e_flags);

/* display ELF header size in bytes */
printf("ELF header size:                  %d bytes\n", elfhdr->e_ehsize);

/* display program header table entry size */
printf("program header table entry size:  %d bytes\n", \
elfhdr->e_phentsize);
if (elfhdr->e_phentsize > 0)
    chk_phentsize=1;

/* display program header table entry count */
printf("program header table entry count: %d\n", \
elfhdr->e_phnum);
if (elfhdr->e_phnum > 0)
    chk_phnum=1;

/* display section header table entry size */
printf("section header table entry size:  %d bytes\n", \
elfhdr->e_shentsize);

/* display section header table entry count */
printf("section header table entry count: %d\n", \
elfhdr->e_shnum);

/* display section header string table index */
printf("section header string table index: %d\n", \
elfhdr->e_shstrndx);

/* check if we can work with program headers */
/* for CORE rest will be empty */
/* add other checks here if intentions to work with ELF in general */
/* if value is zero this means that it doesn't hold an entry */
/* unless ... pretty self explanatory */

if (chk_phoff && chk_phentsize && chk_phnum)
{
    /* table headers */
    printf("\n.: SEGMENT INFORMATION :.\n");
    printf("type offset  vaddress paddress ");
    printf("size(file)  size(mem)  RWX  align\n");
    printf("=====");
    printf("=====\n");

    /* loop through all segments */
    for(count=0; count < elfhdr->e_phnum; ++count)
    {
        /* set the offset appropriately */
        proghdr = (Elf32_Phdr *) ((unsigned char *) \
&longarray[(unsigned int) \
(next_off + elfhdr->e_phoff)]);

        /* display all members as specified by table columns */

        /* determine segment type */
        if (proghdr->p_type == PT_NULL)
            printf("NULL");
        else if (proghdr->p_type == PT_LOAD)
            printf("LOAD");
        else if (proghdr->p_type == PT_DYNAMIC)
            printf("DYNAMIC");
        else if (proghdr->p_type == PT_INTERP)
            printf("INTERP");
        else if (proghdr->p_type == PT_NOTE)
        {
            /* if a note then store the offset into array
            later this array will be processed
            */
            printf("NOTE");
            noteoffsets[note_sect_counter]=proghdr->p_offset;
            note_sect_counter++;
        }
        else if (proghdr->p_type == PT_SHLIB)
            printf("SHLIB");
    }
}

```

```

else if (proghdr->p_type == PT_PHDR)
    printf("PHDR");
else if (proghdr->p_type == PT_TLS)
    printf("TLS");
else if (proghdr->p_type == PT_NUM)
    printf("NUM");
else if (proghdr->p_type == PT_LOOS)
    printf("LOOS");
else if (proghdr->p_type == PT_GNU_EH_FRAME)
    printf("GNU_EH_FRAME");
else if (proghdr->p_type == PT_LOSUNW)
    printf("LOSUNW");
else if (proghdr->p_type == PT_SUNWBSS)
    printf("SUNWBSS");
else if (proghdr->p_type == PT_SUNWSTACK)
    printf("SUNWSTACK");
else if (proghdr->p_type == PT_HISUNW)
    printf("HISUNW");
else if (proghdr->p_type == PT_HIOS)
    printf("HIOS");
else if (proghdr->p_type == PT_LOPROC)
    printf("LOPROC");
else if (proghdr->p_type == PT_HIPROC)
    printf("HIPROC");
else
    printf("UNKNOWN");

/* print the rest */
printf(" %#7x ", proghdr->p_offset);
printf(" %#10x ", proghdr->p_vaddr);
printf(" %#9x ", proghdr->p_paddr);
printf(" %11d ", proghdr->p_filesz);
printf(" %12d ", proghdr->p_memsz);
printf(" %#7x ", proghdr->p_flags);
printf(" %#9x ", proghdr->p_align);

/* next row */
printf("\n");

/* increase offset to next program header table entry */
next_off+=elfhdr->e_phentsize;
}

/* will try to make this with pointers */
noteoffsets[note_sect_counter]='\0';
}

/* need to get rid of this declaration and use more dynamic way of obtaining it */
#define NOTE_PADDING 4

/* determine if there are any note sections recorded contained in the program header */
if (note_sect_counter != 0)
{
    /* note sections counter */
    int noteprcnt;

    printf("\n.: NOTES INFORMATION :.\n");
    unsigned int note_next_note_off=0; //next note offset initially 0
    /* go through each note section and display all the notes in it */
    for(noteprcnt=0; noteprcnt< note_sect_counter; ++noteprcnt)
    {
        unsigned int note_sect_off = noteoffsets[noteprcnt];

        printf("\n### found note section at offset: %#x ###\n", note_sect_off);

        int i=0;
        int note_counter=0;

        /* stupid loop this should contain a logical end of the notes in the section */
        while( i < 3 )
        {
            int j=0;
            char *note_name; /* note name storage */
            unsigned int note_begin_off=0; /* beginning of the next note in
the note section */
            unsigned int note_name_begin_off=0; /* location of the note name
member */
            unsigned int note_desc_begin_off=0; /* location of description member
*/

```

```

unsigned int note_part_total=0;          /* size of note header */
unsigned int note_name_total=0;         /* size of name */
unsigned int note_desc_total=0;        /* size of desc */

/* advance to the next note within the section */

//printf("%#x\n", note_next_note_off);
note_begin_off = (unsigned int) note_sect_off + note_next_note_off;
printf("\n--- note %d at offset %#x ---\n", note_counter, note_begin_off);

/*
the structure of a note in an ELF

          0  1  2  3 -th byte
a          |-----|
s          |  namesz  |
c          |-----| appropriate padding is used for first 3 fields
e          |  descsz  | rest are limited by the upper
n          |-----| for 64 bit arch i would assume we will
d          |   type   | have 8 byte padding instead of 4
          |-----|
|          |   name   |
|          |-----|
|          |   desc   |
V          |-----|

*/

/* find the note */
Elf32_Note_Part *elfnote = (Elf32_Note_Part *) ((unsigned char *)
&longarray[note_begin_off]);

/* includes terminating character */
printf("padding:          %d bytes\n", NOTE_PADDING);
printf("note name size:   %#x bytes\n", elfnote->namesz);
printf("note description size: %#x bytes\n", elfnote->descsz);

/* total bytes of the partial elf note */
note_part_total = (unsigned int) (sizeof(elfnote->namesz) + sizeof(elfnote-
>descsz) + sizeof(elfnote->type));

/* calculate the name offset */
note_name_begin_off = note_begin_off + note_part_total;
/* get note name and display it */
note_name = &longarray[note_name_begin_off];
printf ("note name:          %s\n", note_name);

/* padding must be accounted for */
note_name_total = (unsigned int) ( elfnote->namesz + \
(NOTE_PADDING - ((elfnote->namesz) %
NOTE_PADDING)));

/* padding must be accounted for */
/*note_desc_begin_off = note_name_begin_off + (unsigned int) (
strlen(note_name) + 1 + \
NOTE_PADDING));

*/
note_desc_begin_off = note_name_begin_off + note_name_total;

switch (elfnote->type)
{
    case NT_PRSTATUS:
        printf("note type:          PRSTATUS [%d]\n", elfnote-
>type);
        prstatus_bool = TRUE;
        /* now initialize
the prstatus structure */
        struct elf_prstatus *prstat = (struct elf_prstatus *) (
(unsigned char *) &longarray[note_desc_begin_off]);

        /* display members as defined in sys/procfs.h */
        /* signal info */
        printf("signal number:          %d\n", prstat-
>pr_info.si_signo);
        printf("extra code:          %d\n", prstat-
>pr_info.si_code);
        printf("errno:          %d\n", prstat-
>pr_info.si_errno);

        /* continue with prstatus */
        printf("current signal:          %d\n", prstat->pr_cursig);

```

```

>pr_sigpend);
>pr_sigpend);

>pr_utime.tv_sec, prstat->pr_utime.tv_usec);
>pr_stime.tv_sec, prstat->pr_stime.tv_usec);
>pr_cutime.tv_sec, prstat->pr_cutime.tv_usec);
>pr_cstime.tv_sec, prstat->pr_cstime.tv_usec);

being used */

long int for some reason */
this??? */

char *) prstat->pr_reg);

printf("set of pending signals: %#lx\n", prstat->pr_sigpend);
printf("set of held signals:   %#lx\n", prstat->pr_sighold);

printf("pid:                %d\n", prstat->pr_pid);
printf("ppid:               %d\n", prstat->pr_ppid);
printf("pgrp:               %d\n", prstat->pr_pgrp);
printf("sid:                %d\n", prstat->pr_sid);
printf("user time:          %d.%d sec\n", prstat->pr_utime.tv_sec, prstat->pr_utime.tv_usec);

printf("system time:        %d.%d sec\n", prstat->pr_stime.tv_sec, prstat->pr_stime.tv_usec);
printf("cumulative user time: %d.%d sec\n", prstat->pr_cutime.tv_sec, prstat->pr_cutime.tv_usec);
printf("cumulative system time: %d.%d sec\n", prstat->pr_cstime.tv_sec, prstat->pr_cstime.tv_usec);

/* print out boolean that indicates TRUE if math copro
printf("bool pr_fpvalid:      %d\n", prstat->pr_fpvalid);

/* display GP (general purpose) registers' values */
/* the user_regs struct has been converted as an array of
/* size of this array is stored in ELF_NGREG ...why all
/* let's change it */
user_regs_struct *u_regs = (user_regs_struct *) ((unsigned

/* print the values */
printf("\ngeneral purpose registers:\n");
printf("ebx: %#10lx\t\t", u_regs->ebx);
printf("ecx: %#10lx\t\t", u_regs->ecx);
printf("edx: %#10lx\n", u_regs->edx);
printf("esi: %#10lx\t\t", u_regs->esi);
printf("edi: %#10lx\t\t", u_regs->edi);
printf("ebp: %#10lx\n", u_regs->ebp);
printf("eax: %#10lx\t\t", u_regs->eax);
printf("xds: %#10lx\t\t", u_regs->xds);
printf("xes: %#10lx\n", u_regs->xes);
printf("xfs: %#10lx\t\t", u_regs->xfs);
printf("xgs: %#10lx\t\t", u_regs->xgs);
printf("orig_eax: %#10lx\n", u_regs->orig_eax);
printf("eip: %#10lx\t\t", u_regs->eip);
printf("xcs: %#10lx\t\t", u_regs->xcs);
printf("eflags: %#12lx\n", u_regs->eflags);
printf("esp: %#10lx\t\t", u_regs->esp);
printf("xss: %#10lx\t\t", u_regs->xss);

note_desc_total = (unsigned int) sizeof(struct
note_next_note_off = (unsigned int) note_next_note_off +
(note_part_total + note_name_total + note_desc_total);
/*end of prstatus note handling section */
break;
case NT_FPREGSET:
printf("note type:          FPREGSET [%d]\n", elfnote->type);
break;
case NT_PRPSINFO:
printf("note type:          PRPSINFO [%d]\n", elfnote->type);

prpsinfo_bool = TRUE;
struct elf_prpsinfo *prpsinfo = (struct elf_prpsinfo *) (
(unsigned char *) &longarray[note_desc_begin_off]);

printf("numeric process state: %c\n", prpsinfo->pr_state);
printf("char for pr_state:    %c\n", prpsinfo->pr_sname);
printf("zombie:              %c\n", prpsinfo->pr_zomb);
printf("nice val:            %c\n", prpsinfo->pr_nice);
printf("flag 1:              %#lx\n", prpsinfo->pr_flag1);

printf("uid:                %d\n", prpsinfo->pr_uid);
printf("gid:                %d\n", prpsinfo->pr_gid);
printf("pid:                %d\n", prpsinfo->pr_pid);
printf("ppid:               %d\n", prpsinfo->pr_ppid);
printf("pgrp:               %d\n", prpsinfo->pr_pgrp);
printf("sid:                %d\n", prpsinfo->pr_sid);
printf("filename or executable: %s\n", prpsinfo->pr_fname);
//for (j=0; j < ELF_PRARGSZ; ++j)

```

```

//{
//      printf("argument list : %c\n", prpsinfo-
>pr_psargs[i]);
//}

elf_prpsinfo);
note_desc_total = (unsigned int) sizeof(struct
(note_part_total + note_name_total + note_desc_total);
note_next_note_off = (unsigned int) note_next_note_off +
break;

case NT_PRXREG:
printf("note type:          PRXREG [%d]\n", elfnote-
>type);
note_desc_total = elfnote->descsz;
//next note offset for now
note_next_note_off = (unsigned int) note_next_note_off +
note_part_total + note_name_total + note_desc_total;
printf("next note at %#x\n", note_next_note_off);
break;
//for completeness
/*case NT_TASKSTRUCT:
printf("note type:          TASKSTRUCT [%d]\n",
elfnote->type);
break;
*/
case NT_PLATFORM:
printf("note type:          PLATFORM [%d]\n", elfnote-
>type);
break;
case NT_AUXV:
printf("note type:          AUXV [%d]\n", elfnote-
>type);
break;
case NT_GWINDOWS:
printf("note type:          GWINDOWS [%d]\n", elfnote-
>type);
break;
case NT_ASRS:
printf("note type:          ASRS [%d]\n", elfnote-
>type);
break;
case NT_PSTATUS:
printf("note type:          PSTATUS [%d]\n", elfnote-
>type);
break;
case NT_PSINFO:
printf("note type:          PSINFO [%d]\n", elfnote-
>type);
break;
case NT_PRCRED:
printf("note type:          PRCRED [%d]\n", elfnote-
>type);
break;
case NT_UTSNAME:
printf("note type:          UTSNAME [%d]\n", elfnote-
>type);
break;
case NT_LWPSTATUS:
printf("note type:          LWPSTATUS [%d]\n", elfnote-
>type);
break;
case NT_LWPSINFO:
printf("note type:          LWPSINFO [%d]\n", elfnote-
>type);
break;
case NT_PRFPXREG:
printf("note type:          PRFPXREG [%d]\n", elfnote-
>type);
break;
default:
printf("note type:          not recognized [%d]\n",
elfnote->type);
break;
}

++i;
++note_counter;

```

```

    }
}
/*
debugging
    dump file contents onto stdout remove comment if necessary ... just for
    write(fileno(stdout), longarray, statbuf.st_size);
*/

/* Section header processing */
}

/* Section header processing */
elfhdr1 = (Elf32_Ehdr *) ((unsigned char *) &longarray[0x1000]);
printf("\n.: ENCLOSED ELF HEADER INFORMATION at 0x1000 :.\n");
/* determine file class */
if (elfhdr1->e_ident[EI_CLASS] == ELFCLASS32)
    printf("capacity:                32 bit (ELFCLASS32)  [%d]\n", \
           elfhdr1->e_ident[EI_CLASS]);
else if (elfhdr1->e_ident[EI_CLASS] == ELFCLASS64)
    printf("capacity:                64 bit (ELFCLASS64)  [%d]\n", \
           elfhdr1->e_ident[EI_CLASS]);
else if (elfhdr1->e_ident[EI_CLASS] == ELFCLASSNONE)
    printf("capacity:                none (ELFCLASSNONE) [%d]\n", \
           elfhdr1->e_ident[EI_CLASS]);
else
    printf("error:                    capacity not defined within ELF header [%d]\n", \
           elfhdr1->e_ident[EI_CLASS]);

/* determine data encoding type */
if (elfhdr1->e_ident[EI_DATA] == ELFDATA2LSB)
    printf("data encoding type:          2's comp, little endian (ELFDATA2LSB) [%d]\n", \
           elfhdr1->e_ident[EI_DATA]);
else if (elfhdr1->e_ident[EI_DATA] == ELFDATA2MSB)
    printf("data encoding type:          2's comp, big endian (ELFDATA2MSB) [%d]\n", \
           elfhdr1->e_ident[EI_DATA]);
else if (elfhdr1->e_ident[EI_DATA] == ELFDATANONE)
    printf("data encoding type:          none (ELFCLASSNONE) [%d]\n", \
           elfhdr1->e_ident[EI_DATA]);
else
    printf("data encoding type:          not defined within ELF header [%d]\n", \
           elfhdr1->e_ident[EI_DATA]);

/* determine version number (1 is current) and this is a must */
if (elfhdr1->e_ident[EI_VERSION] == EV_CURRENT)
    printf("file version:                current (EV_CURRENT) [%d]\n", \
           elfhdr1->e_ident[EI_VERSION]);
else
    printf("file version:                illegal version [%d]\n", \
           elfhdr1->e_ident[EI_VERSION]);

/* determine OS ABI identification (application binary interface) */
if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_NONE || elfhdr1->e_ident[EI_OSABI] == ELFOSABI_SYSV)
    printf("OS ABI:                        Unix System V ABI [%d]\n", \
           elfhdr1->e_ident[EI_OSABI]);
else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_HPUX)
    printf("OS ABI:                        HP-UX (ELFOSABI_HPUX) [%d]\n", \
           elfhdr1->e_ident[EI_OSABI]);
else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_NETBSD)
    printf("OS ABI:                        NetBSD (ELFOSABI_NETBSD) [%d]\n", \
           elfhdr1->e_ident[EI_OSABI]);
else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_LINUX)
    printf("OS ABI:                        Linux (ELFOSABI_LINUX) [%d]\n", \
           elfhdr1->e_ident[EI_OSABI]);
else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_SOLARIS)
    printf("OS ABI:                        Solaris (ELFOSABI_SOLARIS) [%d]\n", \
           elfhdr1->e_ident[EI_OSABI]);
else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_AIX)
    printf("OS ABI:                        AIX (ELFOSABI_AIX) [%d]\n", \
           elfhdr1->e_ident[EI_OSABI]);
else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_IRIX)
    printf("OS ABI:                        Irix (ELFOSABI_IRIX) [%d]\n", \
           elfhdr1->e_ident[EI_OSABI]);
else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_FREEBSD)
    printf("OS ABI:                        FreeBSD (ELFOSABI_FREEBSD) [%d]\n", \
           elfhdr1->e_ident[EI_OSABI]);
else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_TRU64)

```

```

        printf("OS ABI:                               Compaq TRU64 Unix (ELFOSABI_HPUX) [%d]\n", \
               elfhdr1->e_ident[EI_OSABI]);
    else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_MODESTO)
        printf("OS ABI:                               Novell Modesto (ELFOSABI_MODESTO) [%d]\n", \
               elfhdr1->e_ident[EI_OSABI]);
    else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_OPENBSD)
        printf("OS ABI:                               OpenBSD          (ELFOSABI_OPENBSD) [%d]\n", \
               elfhdr1->e_ident[EI_OSABI]);
    else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_ARM)
        printf("OS ABI:                               ARM            (ELFOSABI_ARM)      [%d]\n", \
               elfhdr1->e_ident[EI_OSABI]);
    else if (elfhdr1->e_ident[EI_OSABI] == ELFOSABI_STANDALONE)
        printf("OS ABI:                               Standalone (Embedded) Application\n", \
               elfhdr1->e_ident[EI_OSABI]);
    (ELFOSABI_STANDALONE) [%d]\n", \
        elfhdr1->e_ident[EI_OSABI]);
    else
    {
        printf("OS ABI: unknown                               [%d]\n", elfhdr1->e_ident[EI_OSABI]);
        os_abi_unknown = TRUE;
    }

    /* determine ABI version if aplicable */
    if (!os_abi_unknown)
        printf("ABI version:                               %d\n", elfhdr1->e_ident[EI_ABIVERSION]);
    else
        printf("ABI version:                               unknown          [%d]\n", elfhdr1->
>e_ident[EI_ABIVERSION]);

    /* display number of padding bytes */
    printf("padding bytes:                               %d\n", elfhdr1->e_ident[EI_PAD]);

    /* determine object file type */
    if (elfhdr1->e_type == ET_NONE)
        printf("object file type:                               no file type    (ET_NONE) [%d]\n", \
               elfhdr1->e_type);
    else if (elfhdr1->e_type == ET_REL)
        printf("object file type:                               relocatable file (ET_REL) [%d]\n", \
               elfhdr1->e_type);
    else if (elfhdr1->e_type == ET_EXEC)
        printf("object file type:                               executable file (ET_EXEC) [%d]\n", \
               elfhdr1->e_type);
    else if (elfhdr1->e_type == ET_DYN)
        printf("object file type:                               shared object file (ET_DYN) [%d]\n", \
               elfhdr1->e_type);
    else if (elfhdr1->e_type == ET_CORE)
        printf("object file type:                               core file (ET_CORE) [%d]\n", \
               elfhdr1->e_type);
    else if (elfhdr1->e_type == ET_NUM)
        printf("object file type:                               number of defined types (ET_NUM) [%d]\n", \
               elfhdr1->e_type);
    else if (elfhdr1->e_type == ET_LOOS)
        printf("object file type:                               OS-specific range start (ET_LOOS) [%4x]\n", \
               elfhdr1->e_type);
    else if (elfhdr1->e_type == ET_HIOS)
        printf("object file type:                               OS-specific range end (ET_HIOS) [%4x]\n", \
               elfhdr1->e_type);
    else if (elfhdr1->e_type == ET_LOPROC)
        printf("object file type:                               Processor-specific range start (ET_LOPROC) [%4x]\n", \
               elfhdr1->e_type);
    else if (elfhdr1->e_type == ET_HIPROC)
        printf("object file type:                               Processor-specific range end (ET_HIPROC) [%4x]\n", \
               elfhdr1->e_type);
    else
        printf("object file type:                               undefined [%4x]\n", elfhdr1->e_type);

    /* determine architecture */
    /* 100 definitions enter here... just 3 for right now*/

    if (elfhdr1->e_machine == EM_NONE)
        printf("architecture:                               none          (EM_NONE) [%d]\n", \
               elfhdr1->e_machine);
    else if (elfhdr1->e_machine == EM_386)
        printf("architecture:                               Intel 80386 (EM_NONE) [%d]\n", \
               elfhdr1->e_machine);
    else if (elfhdr1->e_machine == EM_SPARC)
        printf("architecture:                               Sun SPARC (EM_SPARC) [%d]\n", \
               elfhdr1->e_machine);
    else
        printf("architecture:                               unknown [%d]\n", elfhdr1->e_machine);

```

```

/* determine object file version */
if (elfhdr1->e_version == EV_NONE)
printf("object file version:                invalid ELF version (EV_NONE) [%d]\n", \
      elfhdr1->e_version);
else if (elfhdr1->e_version == EV_CURRENT)
printf("object file version:                current (EV_CURRENT) [%d]\n", \
      elfhdr1->e_version);
else printf("object file version:          unknown [%d]\n", elfhdr1->e_version);

/* display the entry point virtual address */
printf("entry point virtual address:        %#x\n", elfhdr1->e_entry);

/* display the program header table file offset */
printf("program header table file offset:   %#x\n", elfhdr1->e_phoff);
if (elfhdr1->e_phoff > 0)
    chk_phoff=1;

/* display section header table file offset */
printf("section header table file offset:   %#x\n", elfhdr1->e_shoff);

/* display processor-specific flags */
printf("processor specific flags:           %#x\n", elfhdr1->e_flags);

/* display ELF header size in bytes */
printf("ELF header size:                    %d bytes\n", elfhdr1->e_ehsize);

/* display program header table entry size */
printf("program header table entry size:    %d bytes\n", \
      elfhdr1->e_phentsize);
if (elfhdr1->e_phentsize > 0)
    chk_phentsize=1;

/* display program header table entry count */
printf("program header table entry count:   %d\n", \
      elfhdr1->e_phnum);
if (elfhdr1->e_phnum > 0)
    chk_phnum=1;

/* display section header table entry size */
printf("section header table entry size:    %d bytes\n", \
      elfhdr1->e_shentsize);

/* display section header table entry count */
printf("section header table entry count:   %d\n", \
      elfhdr1->e_shnum);

/* display section header string table index */
printf("section header string table index:  %d\n", \
      elfhdr1->e_shstrndx);

/* unmap file; free the memory */
munmap(&longarray, statbuf.st_size);

return(1);
}

```