Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

Ltrace Internals

Rodrigo Rubira Branco
IBM
rrbranco@br.ibm.com

Abstract

ltrace is a program that permits you to track runtime library calls in dynamically linked programs without recompiling them, and is a really important tool in the debugging arsenal. This article will focus in how it has been implemented and how it works, trying to cover the actual lacks in academic and in-deep documentation of how this kind of tool works (setting the breakpoints, analysing the executable/library symbols, interpreting elf, others).

1 Introduction

ltrace is divided into many source files; some of these contain architecture-dependent code, while some others are generic implementations.

The idea is to go through the functions, explaining what each is doing and how it works, beginning from the entry point function, `main`.

2 int main(int argc, char **argv) – ltrace.c

The `main` function sets up ltrace to perform the rest of its activities.

It first sets up the terminal using the `guess_cols()` function that tries to ascertain the number of columns in the terminal so as to display the information output by ltrace in an ordely manner. The column count is initially queried from the `$COLUMNS` environment variable (if that is not set, the TIOCGWINSZ ioctl is used instead). Then the program options are handled using the `process_options()` function to processes the ltrace command line arguments, using the `getopt()` and `getopt_long()` functions to parse them.

It then calls the `read_config_file()` function on two possible configuration files.

It calls `read_config_file()` first with SYSCONFDIR's `ltrace.conf` file. If `$HOME` is set, it then calls the function with `$HOME/.ltrace.conf`. This function opens the specified file and reads in from it line-by-line, sending each line to the `process_line()` function to verify the syntax of the config file based on the line supplied to it. It then returns a function structure based on the function information obtained from said line.

If `opt_e` is set, then a list is output by the `debug()` function.

If passed a command invocation, `ltrace` will execute it via the `execute_program()` function which takes the return value of the `open_program()` function as an argument.

Ltrace will attached to any supplied pids using the `open_pid()` function.

At the end of this function the `process_event()` function is called in an infinite loop, receiving the return value of the `wait_for_something()` function as its argument.

3 struct process *open_program(char *filename, pid_t pid) – proc.c

This function implements a number of important tasks needed by ltrace. `open_program` allocates a process structure's memory and sets the filename and pid (if needed), adds the process to the linked-list of processes traced by ltrace, and most importantly initalizes breakpoints by calling `breakpoints_init()`.

4 void breakpoints_init(struct process *proc) – breakpoints.c

The `breakpoints_init()` function is responsible for setting breakpoints on every symbol in the program being traced. It calls the `read_elf()` function which returns an array of library_symbol structures, which it processes based on opt_e. Then it iterates through the array of library_symbol structures and calls the insert_breakpoint() function on each symbol.

5 struct library_symbol *read_elf(struct process *proc) – elf.c

This function retrieves a process's list of symbols to be traced. It calls do_init_elf() on the executable name of the traced process and for each library supplied by the -l option. It loops across the PLT information found therein.

For each symbol in the PLT information, a GElf_Rel structure is returned by a call to gelf_getrel(), if the d_type is ELF_T_REL and gelf_getrela() if not. If the return value of this call is NULL, or if the value returned by ELF64_R_SYM(rela.r_info) is greater than the number of dynamic symbols or the rela.r_info symbol is not found, then the function calls the error() function to exit the program with an error.

If the symbol value is NULL and the PLTs_initialized_by_here flag is set, then the need_to_reinitialize_breakpoints member of the proc structure is set.

The name of the symbol is calculated and this is passed to a call to in_load_libraries(). If this returns a positive value, then the symbol address is calculated via the arch_plt_sym_val() function and the add_library_symbol() function is called to add the symbol to the library_symbols list of dynamic symbols. At this point if the need_to_reinitialize_breakpoints member of the proc structure is set, then a pt_e_t structure main_cheat is allocated and its values are set. After this a loop is made over the opt_x value (passed by the -x option) and if the PLTs_initialized_by_here variable matches the name of one of the values, then main_cheat is freed and the loop is broken. If no match is found, then opt_x is set to the final value of main_cheat.

A loop is then made over the symtab, or symbol table variable. For each symbol gelf_getsym() is called, which if it fails provokes ltrace to exit with an error message via the error() function. A nested loop is then made over the values passed to opt_x via the -x option. For each value a comparison is made against the name of each symbol. If there is a match, then the symbol is added to the library_symbols list via add_library_symbol() and the nested loop breaks.

At the end of this loop a final loop is made over the values passed to opt_x via the -x option.

For each value with a valid name member a comparison is made to the E_ENTRY_NAME value, which represents the program's entry point. If this comparison should prove true, then the symbol is entered into the library_symbols list via add_library_symbol().

At the end of the function, any libraries passed to ltrace via the -l option are closed via the do_close_elf() function[1] and the library_symbols list is returned.

6 static void do_init_elf(struct ltelf *lte, const char *filename) – elf.c

The passed ltelf structure is set to zero and open() is called to open the passed filename as a file. If this fails, then ltrace exits with an error message. The elf_begin() function is then called, following which various checks are made via elf_kind() and gelf_getehdr(). The type of the elf header is checked so as to only process executable files or dynamic library files.

If the file is not of one of these types, then ltrace exits with an error. Ltrace also exits with an error if the elf binary is from an unsupported architecture.

The ELF section headers are iterated over and the elf_getscn() function is called, then the variable *name* is set via the elf_strptr() function (if any of the above functions fail, ltrace exits with an error message).

A comparison is then made against the section header type and the data for it is obtained via a call to elf_getdata().

[1] This function is called to close open ELF images. A check is made to see if the ltelf structure has an associated hash value allocated and if so this hash value is deallocated via a call to free(). After this elf_end() is called and the file descriptor associated with the image is closed.

For SHT_DYNSYM (dynamic symbols), the lte->dynsym is filled via a call to elf_getdata(), where the dynsym_count is calcuated by dividing the section header size by the size of each entry. If the attempt to get the dynamic symbol data fails, ltrace exits with an error message. The elf_getscn() function is then called, passing the section header sh_link variable. If this fails, then ltrace exits with an error message. Using the value returned by elf_getscn(), the gelf_getshdr() function is called and if this fails, ltrace exits with an error message.

For SHT_DYNAMIC an Elf_Data structure *data* is set via a call to elf_getdata() and if this fails, ltrace exits with an error message. Every entry in the section header is iterated over and the following occurs: The gelf_getdyn() function is called to retrieve the .dynamic data and if this fails, ltrace exits with an error message; relplt_addr and relplt_size are calculated from the returned dynamic data.

For SHT_HASH values an Elf_Data structure *data* is set via a call to elf_getdata() and if this fails, ltrace exits with an error message. If the entry size is 4 then lte->hash is simply set to the dynamic data buf data->d_buf. Otherwise it is 8. The correct amount of memory is allocated via a call to malloc and the hash data into copied into lte->hash.

For SHT_PROGBITS, checks are made to see if the name value is .plt or .pd, and if so, the correct elements are set in the lte->plt_addr/lte->opd and lte->plt_size and lte->pod_size structures. In the case of OPD, the lpe->opd structure is set via a call to elf_rawdata(). If neither the dynamic symbols or the dynamic strings have been found, then ltrace exits with an error message. If relplt_addr and lte->plt_addr are non-null, the section headers are iterated across and the following occurs:

- The elf_getscn() function is called.
- If the sh_addr is equal to the relpt_addr and the sh_size matches the relplt_size (i.e., this section is the .relplt section) then lte->relplt is obtained via a call to elf_getdata() and lte->relplt_count is calculated as the size of section divided by the size of each entry. If the call to elf_getdata() fails then ltrace exits with an error message.
- If the function was unable to find the .relplt section then ltrace exits with an error message.

7 static void add_library_symbol(GElf_Addr addr, const char *name, struct library_symbol **library_symbolspp, int use_elf_plt2addr, int is_weak) – elf.c

This function allocates a library_symbol structure and inserts it into the linked list of symbols represented by the library_symbolspp variable.

The structure is allocated with a call to malloc(). The elements of this structure are then set based on the arguments passed to the function. And the structure is linked into the linked list using its *next* element.

8 static GElf_Addr elf_plt2addr(struct ltelf *lte, void *addr) – elf.c

In this function the opd member of the lte structure is checked and if it is NULL, the function returns the passed address argument as the return value. If opd is non-NULL, then following occurs:

1. An offset value is calculated by subtracting the opd_addr element of the ltr structure from the passed address.
2. If this offset is greater than the opd_size element of the lte structure then ltrace exits with an error.
3. The return value is calculated as the base address (passed as lte->opd->d_buf) plus the calculated offset value.
4. This calculated final return value is returned as a GElf_Addr variable.

9 static int in_load_libraries(const char *name, struct ltelf *lte) – elf.c

This functions checks if there are any libraries passed to ltrace as arguments to the -l option. If not, then the function immediately returns 1 (one) because there is no filtering (specified libraries) in place; otherwise, a hash is calculated for the library name arguments by way of the elf_hash() function.

For each library argument, the following occurs:

1. If the hash for this iteration is NULL the loop continues to the next iteration.

2. The *nbuckets* value is obtained and the buckets and chain values are calculated based on this value from the hash.

3. For each bucket the following occurs:

   The `gelf_getsym()` function is called to get the symbol; if this fails, then ltrace exits with an error.

   A comparison is made between the passed name and the name of the current dynamic symbol. Should there be a match, the function will return a positive value (one).

4. If the code reaches here, 0 (zero) is returned.

10 void insert_breakpoint(struct process *proc, void *addr, struct library_symbol *libsym) – breakpoints.c

The `insert_breakpoint()` function inserts a breakpoint into a process at the given address (addr). If the breakpoints element of the passed proc structure has not been set it is set by calling the `dict_init()` function.

A search is then made for the address by using the `dict_find_entry()` function. If the address is not found a breakpoint structure is allocated using `calloc()`, entered into the dict hash table using `dict_enter()`, and its elements are set.

If a pid has been passed (indicating that the process is already running), this breakpoint structure along with the pid is then passed to the `enable_breakpoint()` system-dependent function.

11 void enable_breakpoint(pid_t pid, struct breakpoint *sbp) – sysdeps/linux-gnu/breakpoint.c

The `enable_breakpoint()` function is responsible for the insertion of breakpoints into a running process using the ptrace interface.

First `PTRACE_PEEKTEXT` ptrace parameter is used to save the original data from the breakpoint location and then `PTRACE_POKETEXT` is used to copy the architecture-dependent breakpoint value into the supplied memory address. The architecture-dependent breakpoint value is found in `sysdeps/linux-gnu/*/arch.h`.

12 void execute_program(struct process *sp, char **argv) – execute-program.c

The `execute_program()` function executes a program whose name is supplied as an argument to ltrace. It `fork()`s a child, changes the UID of the running child process if necessary, calls the `trace_me()` (simply calls ptrace() using the `PTRACE_TRACEME` argument, which allows the process to be traced) function and then executes the program using `execvp()`.

13 struct event *wait_for_something(void) – wait_for_something.c

The `wait_for_something()` function literally waits for an event to occur and then handles it.

The events that it treats are: Syscalls, Systets, Exiosts, exit signals, and breakpoints. `wait_for_something()` calls the `wait()` function to wait for an event.

When it awakens it calls `get_arch_dep()` on the proc member of the event structure. If breakpoints were not enabled earlier (due to the process not yet being run) they are enabled by calling `enable_all_breakpoints()`, `trace_set_options()` and then `continue_process()` (this function simply calls `continue_after_signal()`).

In this case the event is then returned as `LT_EV_NONE` which does not receive processing.

To determine the type of event that has occurred the following algorithm is used: The `syscall_p()` function is called to detect if a syscall has been called via int 0x80 (`LT_EV_SYSCALL`) or if there has been a return-from-syscall event (`LT_EV_SYSRET`). If neither of these is true, it checks to see if the process has exited or has sent an exit signal.

If neither of these is the case and the process has not stopped, an `LT_EV_UNKNOWN` event is returned.

If process is stopped and the stop signal was not `systrap`, an `LT_EV_SIGNAL` event is returned.

If none of the above cases is found to be true, it is assumed that this was a breakpoint, and an `LT_EV_BREAKPOINT` event is returned.

14 void process_event(struct event *event) – process_event.c

The process_event() function receives an event structure, which is generally returned by the wait_for_something() function.

It calls a switch-case construct based on the event->thing element and processes the event using one of the following functions: process_signal(), process_exit(), process_exit_signal(), process_syscall(), process_sysret(), or process_breakpoint().

In the case of syscall() or sysret(), it calls the sysname() function.

15 int syscall_p(struct process *proc, int status, int *sysnum) – sysdeps/linux-gnu/*/trace.c

This function detects if a call to or return from a system call occurred. It does this first by checking the value of EAX (on x86 platforms) which it obtains with a ptrace PTRACE_PEEKUSER operation.

It then checks the program's call stack, as maintained by ltrace and, checking the last stack frame, it sees if the is_syscall element of the proc structure is set, which indicates a called system call. If this is set, then *2* is returned, which indicates a sysret event. If not, then *1* is returned, provided that there was a value in EAX.

16 static void process_signal(struct event *event) – process_event.c

This function tests the signal. If the signal is SIGSTOP it calls disable_all_breakpoints(), untrace_pid() (this function merely calls the ptrace interface using a PTRACE_DETACH operation), removes the process from the list of traced processes using the remove_proc() function, and then calls continue_after_signal() (this function simply calls ptrace with a PTRACE_SYSCALL operation) to allow the process to continue.

In the case that signal was not SIGSTOP, the function calls the output_line() function to display the fact of the signal and then calls continue_after_signal() to allow the process to continue.

17 static void process_exit(struct event *event) – process_event.c

This function is called when a traced process exits. It simply calls output_line() to display that fact in the terminal and then calls remove_proc() to remove the process from the list of traced processes.

18 static void process_exit_signal(struct event *event) – process_event.c

This function is called when when a traced program is killed. It simply calls output_line() to display that fact in the terminal and then calls remove_proc() to remove the process from the list of traced processes.

19 static void process_syscall(struct event *event) – process_event.c

This function is called when a traced program invokes a system call. If the -S option has been used to run ltrace, then the output_left() function is called to display the syscall invocation using the sysname() function to find the name of the system call.

It checks if the system call will result in a fork or execute operation, using the fork_p() and exec_p() functions which test the system call against those known to trigger this behavior. If it is such a signal the disable_all_breakpoints() function is called.

After this callstack_push_syscall() is called, followed by continue_process().

20 static void process_sysret(struct event *event) – process_event.c

This function is called when the traced program returns from a system call. If ltrace was invoked with the -c or -T options, the calc_time_spent() function is called to calculate the amount of time that was spent inside the system call.

After this the function fork_p() is called to test if the system call was one that would have caused a process fork. If this is true, and the -f option was set when running ltrace, then the gimme_arg() function is called to get the pid of the child and the open_pid() function is called to begin tracing the child. In any case, enable_all_breakpoints() is called.

Following this, the callstack_pop() function is called. Then the exec_p() function tests if the system call was one that would have executed another program within this process and if true, the gimme_arg() function is called. Otherwise the event->proc structure is re-initialized with the values of the new program and the breakpoints_init() function is called to initialize breakpoints. If gimme_arg() does not return zero, the enable_all_breakpoints() function is called.

At the end of the function the continue_process() function is called.

21 static void process_breakpoint(struct event *event) – process_event.c

This function is called when the traced program hits a breakpoint, or when entering or returning from a library function.

It checks the value of the event->proc->breakpoint_being_enabled variable to determine if the breakpoint is in the middle of being enabled, in which case it calls the continue_enabling_breakpoint() function and this function returns. Otherwise this function continues.

It then begins a loop through the traced program's call stack, checking if the address where the breakpoint occurred matches a return address of a called function which indicates that the process is returning from a library call.

At this point a hack allows for PPC-specific behavior, and it re-enables the breakpoint. All of the library function addresses are retrieved from the call stack and translated via the plt2addr() function. Provided that the architecture is EM_PPC, the address2bpstruct()[2] function is called to translate the address into a breakpoint structure. The value from the address is read via the ptrace PTRACE_PEEK operation and this value is compared to a breakpoint value. If they do not match, a breakpoint is inserted at the address.

If the architecture is not EM_PPC, then the address is compared against the address of the breakpoint previously applied to the library function. If they do not match, a breakpoint is inserted at the address.

[2] This function merely calls dict_find_entry() to find the correct entry in proc->breakpoints and returns it.

Upon leaving the PPC-dependent hack, the function then loops across callstack frames using the callstack_pop() function until reaching the frame that the library function has returned to which is normally a single callstack frame. Again if the -c or -T options were set, calc_time_spent() is called.

The callstack_pop() function is called one final time to pop the last callstack frame and the process' return address is set in the proc structure as the breakpoint address. The output_right() function is called to log the library call and the continue_after_breakpoint() function is called to allow the process to continue, following which the function returns.

If no return addresses in the callstack match the breakpoint address, the process is executing in, and not returning from a library function.

The address2bpstruct() function is called to translate the address into a breakpoint structure.

Provided that this was a success, the following occurs:

- The stack pointer and return address to be saved in the proc stucture are obtained using the get_stack_pointer() and get_return_address() functions.
- The output_left() function is called to log the library function call and the callstack_push_symfunc() function is called. A check is then made to see if the PLTs_initialized_by_here variable is set, to see if the function matches the called library function's symbol name and to see if the need_to_reinitialize_breakpoints variable is set. If all this is true the reinitialize_breakpoints() function is called.

Finally continue_after_breakpoint() is called and the function returns.

If address2bpstruct() call above was not successful, output_left() is called to show that an unknown and unexpected breakpoint was hit. The continue_process() function is called and the function returns.

22 static void callstack_push_syscall(struct process *proc, int sysnum) – process_event.c

This function simply pushes a `callstack_element` structure onto the array callstack held in the `proc` structure. This structure's `is_syscall` element is set to differentiate this callstack frame from one which represents a library function call. The `proc` structure's member `callstack_depth` is incremented to reflect the callstack's growth.

23 static void callstack_push_symfunc(struct process *proc, struct library_symbol *sym) – process_event.c

As in the `callstack_push_syscall()` function described above, a `callstack_element` structure is pushed onto the array callstack held in the proc structure and the `callstack_depth` element is incremented to reflect this growth.

24 static void callstack_pop(struct process *proc) – process_event.c

This function performs the reverse of the two functions described above. It removes the last structure from the callstack array and decrements the `callstack_depth` element.

25 void enable_all_breakpoints(struct process *proc) – breakpoints.c

This function begins by checking the `breakpoints_enabled` element of the `proc` structure. Only if it is not set the rest of the function continues.

If the architecture is PPC and the option `-L` was **not** used, the function checks if the PLT has been set up by using a ptrace `PTRACE_PEEKTEXT` operation. If not, the function returns at this point.

If `proc->breakpoints` is set the `dict_apply_to_all()` function is called using `enable_bp_cb()` function.[3] This call will set the `proc->breakpoints_enabled`.

[3]This function is a callback that simply calls the function `enable_breakpoint()`.

26 void disable_all_breakpoints(struct process *proc) – breakpoints.c

If `proc->breakpoints_enabled` is set, this function calls `dict_apply_to_all()` with the argument `disable_bp_cb()` as the callback function. It then sets `proc->breakpoints_enabled` to zero and returns.

27 static void disable_bp_cb(void *addr, void *sbp, void *proc) – breakpoints.c

This function is a callback called by `dict_apply_to_all()` and simply calls the function `disable_breakpoint()` (does the reverse of `enable_breakpoint`, copying the saved data from the breakpoint location back over the breakpoint instruction using the ptrace `PTRACE_POKETEXT` interface).

28 void reinitialize_breakpoints(struct process *proc) – breakpoints.c

This function retrieves the list of symbols as a `library_symbol` linked-list structure from the `proc->list_ofsymbols` and iterates over this list, checking each symbol's `need_init` element and calling `insert_breakpoint()` for each symbol for which this is true.

If `need_init` is still set after `insert_breakpoint` an error condition occurs, the error is reported and ltrace exits.

29 void continue_after_breakpoint(struct process *proc, struct breakpoint *sbp) – sysdeps/linux-gnu/trace.c

A check is made to see if the breakpoint is enabled via the `sbp->enabled` flag. If it is then `disable_breakpoint()` is called.

After this, `set_instruction_pointer()`[4] is called to set the instruction pointer to the address of the breakpoint. If the breakpoint is still enabled, then `continue_process()` is called. If not then if the architecture is SPARC or ia64 the `continue_`

[4]This function retrieves the current value of the instruction pointer using the ptrace interface with values of `PTRACE_POKEUSER` and EIP.

process() function is called or if not the ptrace interface is invoked using a PTRACE_SINGLESTEP operation.

30 void open_pid(pid_t pid, int verbose) – proc.c

The trace_pid() function is called on the passed pid, if this fails then the function prints an error message and returns.

The filename for the process is obtained using the pid2name() function and open_program() is called with this filename passed as an argument.

Finally the breakpoints_enabled flag is set in the proc structure returned by open process.

31 static void remove_proc(struct process *proc) – process_event.c

This function removes a process from the linked list of traced processes.

If list_of_processes is equal to proc (i.e., the process was the first in the linked list) then there is a reverse unlink operation where list_of_processes = list_of_processes->next.

If not and the searched-for process is in the middle of the list, then the list is iterated over until the process is found and tmp->next is set to tmp->next->next, simply cutting out the search for process from the linked list.

32 int fork_p(struct process *proc, int sysnum) – sysdeps/linux-gnu/trace.c

This function checks to see if the given sysnum integer refers to a system calls that would cause a child process to be created. It does this by checking the fork_exec_syscalls table using the proc->personality value and an index, *i*, to check each system call in the table sequentially, returning *1* if there is a match.

If the proc->personality value is greater than the size of the table, or should there not be a match, then zero is returned.

33 int exec_p(struct process *proc, int sysnum) – sysdeps/linux-gnu/trace.c

This function checks to see if the given sysnum integer refers to a system calls that would cause another program to be executed. It does this by checking the fork_exec_syscalls table using the proc->personality value and an index, *i*, to check each system call in the table sequentially, returning *1* if there is a match.

If the proc->personality value is greater than the size of the table, or should there not be a match, then zero is returned.

34 void output_line(struct process *proc, char *fmt, ...) – output.c

If the -c option is set, then the function returns immediately. Otherwise the begin_of_line() function[5] is called and the *fmt* argument data is output to the output (can be a file chosen using -o or stderr) using fprintf().

35 void output_left(enum tof type, struct process *proc, char *function_name) – output.c

If the -c option was set, then the function returns immediately. If the current_pid variable is set then the message <unfinished ...> is output and current_pid and current_column are set to zero.

Otherwise current_pid is set to the pid element of the proc structure, and current_depth is set to proc->callstack_depth. The begin_of_line() function is called.

If USER_DEMANGLE is #defined then the function name is output by way of my_demange(), or else it is just output plain.

A variable *func* is assigned by passing the function_name to name2func() if this failed then a loop is iterated four times calling display_arg() many times in succession to display four arguments.

[5] Prints the beginning part of each output line. It prints the process ID, the time passed since the last output line and either the return address of the current function or the instruction pointer.

At the end of the loop it is called a fifth time.

Should the call to name2func() succeed, then another loop is iterated but over the number of parameters that the function receives—for each of which the display_arg() function is called.

Finally if func->params_right is set, save_register_args() is called.

36 void output_right(enum tof type, struct process *proc, char *function_name) – output.c

A function structure is allocated via the name2func() function.

If the -c option was set providing the dict_opt_c variable is not set it is allocated via a call to dict_init(). An opt_c_struct structure is allocated by dict_find_entry(). If this should fail, then the structure is allocated manually by malloc() and the function name is entered into the dictionary using the dict_enter() function.

There are various time calculations and the function returns. If the current_pid is set, is not equal to proc->pid and the current_depth is not equal to the process' callstack_depth then the message <unfinished>... is output and current_pid is set to zero. If current_pid is not equal to the proc structure's pid element then begin_of_line() is called and then if USE_DEMANGLE is defined the function name is output as part of a resumed message using fprintf() via my_demangle(). If USE_DEMANGLE is not defined then fprintf() alone is used to output the message. If *func* is not set then arguments are displayed using ARGTYPE_UNKNOWN, otherwise they are displayed using the correct argument type from the proc structure.

37 int display_arg(enum tof type, struct process *proc, int arg_num, enum arg_type at) – display_args.c

This function displays one of the arguments, the arg_num'th argument to the function the name of which is currently being output to the terminal by the output functions.

It uses a switch case to decide how to display the argument. Void, int, uint, long, ulong, octal char, and address types are displayed using the fprintf() stdio function. String and format types are handled by the display_string, display_stringN() function (sets the string_maxlength by calling gimme_arg() with the *arg2* variable. It then calls display_string()) and display_format() functions respectively.

Unknown values are handled by the display_unknown() function.

38 static int display_unknown(enum tof type, struct process *proc, int arg_num) – display_args.c

The display_unknown() function performs a calculation on the argument, retrieved using the arg_num variable and uses of the gimme_arg() function. Should the value be less than 1,000,000 and greater than –1,000,000 then it is displayed as a decimal integer value; if not, it is interpreted as a pointer.

39 static int display_string(enum tof type, struct process *proc, int arg_num) – display_args.c

The display_string() function uses gimme_arg() function to retrieve the address of the string to be displayed from the stack. If this fails then the function returns and outputs the string **NULL**.

Memory is allocated for the string using malloc() and should this fail, the function returns and outputs **???** to show that the string was unknown.

The umovestr() function is called to copy the string from its address and the length of the string is determined by either the value passed to -s or the maximum length of a string (by default infinite). Each character is displayed by the display_char() function (outputs the supplied character using fprintf(). It converts all the control characters such as \r (carriage return), \n (newline), and EOF (end of file) to printable versions).

Should the string be longer than the imposed maximum string length, then the string "..." is output to show that there was more data to be shown.

The function returns the length of the output.

40 static char *sysname(struct process *proc, int sysnum) – process_event.c

This function retrieves the name of a system call based on its system call number.

It checks the personality element of the proc structure and the sysnum values to check that they fit within the size of the `syscalents[]` array.

If `proc->personality` does not, the `abort()` function is called. If sysnum does not then a string value of `SYS_<sysnum>` is returned.

Provided that both numbers fit within the syscalents array the correct value is obtained using the sysnum variable. A string value of `SYS_<name of systemcall>` is returned.

41 long gimme_arg(enum tof type, struct process *proc, int arg_num) – sysdeps/linux-gnu/*/trace.c

For x86 architecture this function checks if `arg_num` is –1, if so then the value of the EAX register is returned, which is obtained via the ptrace `PTRACE_PEEKUSER` operation.

If type is equal to `LT_TOF_FUNCTION` or `LT_TOF_FUNCTIONR` then the `arg_num`'th argument is returned from the stack via a ptrace `PTRACE_PEEKUSER` operation based on the current stack pointer (from the `proc` structure) and the argument number.

If the type is `LT_TOF_SYSCALL` or `LT_TOF_SYSCALLR` then a register value is returned based on the argument number as so: 0 for EBX, 1 for ECX, 2 for EDX, 3 for ESI, and 4 for EDI.

If the `arg_num` does not match one of the above or the type value does not match either of the above cases, ltrace exits with an error message.

42 static void calc_time_spent(struct process *proc) – process_event.c

This function calculates the time spent in a system call or library function. It retrieves a `callstack_element` structure from the current frame of the process' callstack and calls `gettimeofday()` to obtain the current time and compares the saved time in the `callstack_element` structure to the current time. This difference is then stored in the `current_diff` variable.

43 void *get_instruction_pointer(struct process *proc) – sysdeps/linux-gnu/*/regs.c

This function retrieves the current value of the instruction pointer using the ptrace interface with values of `PTRACE_PEEKUSER` and EIP.

44 void *get_stack_pointer(struct process *proc) – sysdeps/linux-gnu/*/regs.c

This function retrieves the stack pointer of the traced process by using the ptrace interface with values of `PTRACE_PEEKUSER` and UESP.

45 void *get_return_addr(struct process *proc, void *stack_pointer) – sysdeps/linux-gnu/*/regs.c

This function retrieves the current return address of the current stack frame using the ptrace interface `PTRACE_PEEKTEXT` operation to retrieve the value from the memory pointed to by the current stack pointer.

46 struct dict *dict_init(unsigned int (*key2hash) (void *), int (*key_cmp) (void *, void *)) – dict.c

A dict structure is allocated using `malloc()`, following which the buckets array of this structure is iterated over and each element of the array is set to NULL.

The `key2hash` and `key_cmp` elements of the `dict` structure are set to the representative arguments passed to the function and the function returns.

47 int dict_enter(struct dict *d, void *key, void *value) – dict.c

This function enters a value into the linked list represented by the `dict` structure passed as the first argument.

A hash is calculated by the `key2hash()` function using the key argument to the function and a `dict` structure `new_entry`, which is allocated with `malloc()`. The elements of `new_entry` are set using *key*, *value*, and *hash*.

An index is calculated by rounding the hash value with the size of the `d->bucket` array, and the `new_entry` structure is entered into this array at this index by linking it to the start of the linked list held there.

48 void dict_clear(struct dict *d) – dict.c

This function iterates over both the `d->buckets` array and the linked list held in each `d->buckets` array element. For each linked list element it frees the entry before unlinking it from the list. For each emptied bucket it sets the `d->bucket` element to NULL.

49 void *dict_find_entry(struct dict *d, void *key) – dict.c

A hash is created using the `d->key2hash` function pointer and the passed key argument variable.

This hash is then used to index into the `d->buckets` array as a `dict_entry` structure. The linked listed held in this element of the array is iterated over comparing the calculated hash value to the hash value held in each element of the linked list.

Should the hash values match, a comparison is made between the key argument and the key element of this linked list. If this comparison should prove true the function returns the entry. Otherwise the function returns NULL if no matches are ultimately found.

50 void dict_apply_to_all(struct dict *d, void (*func) (void *key, void *value, void *data), void *data) – dict.c

This function iterates over all the elements in the `d->buckets` array, and iterates over the linked list held in each element of said array.

For each element of each linked list the passed func function pointer is called using the key, value and data elements of the supplied dict structure *d*.

51 unsigned int dict_key2hash_string(void *key) – dict.c

This function creates a hash value from a character string passed as the void pointer *key*.

The key is first cast to a character pointer and for each character in this string the following is carried out:

- The integer *total* is incremented by the current value of total XORd by value of the character shifted left by the value shift, which starts out as zero, and is incremented by five for each iteration.
- Should the shift pass the value of 24, it is reduced to zero.

After processing each character in the supplied string the function returns the value held in the variable *total* as the final hash value.

52 dict_key_* helper functions – dict.c

Ltrace have many simple function to help in the key comparisions:

- `int dict_key_cmp_string(void *key1, void *key2) -- dict.c`

  A very simple function that returns the result of a call to `strcmp()` using the two supplied pointer values.

- `unsigned int dict_key2hash_int(void *key) -- dict.c`

  This is a very simple function that returns the supplied pointer value cast to an unsigned int type.

- `int dict_key_cmp_int(void *key1, void *key2) -- dict.c`

  This is a very simple function that returns the mathematical difference of *key2* from *key1*.