Linux Training

Ericsson – Summer 2015

Introduction to operating System concepts using Linux

1 Introduction

The Linux Training is a course where all major operating subsystems are studied, such as the memory management, the scheduling, the system call interface and the interruption processing. The connection is made between the micro-architecture, the operating system and the user's program. It is based on an open source stack, such that the participant can see what is under the hood and get the whole picture. The activities covers both user-space programs and the underlying kernel code. Tracing is used to expose the system behaviour under particular conditions.

1.1 Prerequisite

Here is the list of prerequisite for the activities :

- A recent Ubuntu Linux for compiling and running the activities.
- Eclipse CDT is recommended.
- The archives `linux-training-<version>.tar.gz` and `linux-training-modules-<version>.tar.gz`.
- The Linux kernel source tree
- LTTng kernel and user-space tracer.
- A virtual machine with SSH connection to test kernel modules.
- Required compilation dependencies are specified into README files of the source archive.

1.2 Compiling sources

The linux-training source archive uses autotools and make for building the programs. The following commands decompress and configure the sources.

```
# Uncompress the gzip archive
tar xzvf linux-training-<version>.tar.gz

# Change directory
cd linux-training-<version>/

# Execution the configure script
./configure
```

```
# compile the sources
make
```

The kernel modules are not using autotools. Static makefiles are used instead. By default, the modules will compile using the running kernel headers. The modules are included in the lttng-modules source tree for experimenting with tracing.

```
# Uncompress the gzip archive
tar xzvf linux-training-modules-<version>.tar.gz

# Change directory
cd linux-training-modules-<version>/

# compile the sources
make

# load or reload the modules
./control-addons.sh load
```

An Eclipse CDT project is included in the archive, and can be imported using File → Import… → Existing Projects into Workspace, and then selecting the source directory.

1.3 Linux source tree

Here are the steps to checkout, build the kernel and import the sources in Eclipse.

```
# Checkout the git Linux kernel source tree

git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
cd linux/
git checkout v4.1

# Build the kernel (there are generated files specific to the architecture)
# The localmodconfig target will only select currently loaded modules
# which speed-up the compilation step compared to compile all modules.

make localmodconfig
make -j4
```

There is no need to install the kernel on the host. Installing the kernel is not necessary, because the kernel code for the activities are within modules.

Install the latest Eclipse CDT on your host. It is recommend for browsing the kernel API, and for editing custom modules. Importing sources is different from a typical program. The main difference is related to the include files, because the system includes must not be used, and specific includes for the architecture must be added. Here is the complete procedure to import Linux sources into Eclipse CDT:

https://wiki.eclipse.org/HowTo_use_the_CDT_to_navigate_Linux_kernel_source

Increasing the the JVM max heap is required. Otherwise the indexing will stop. Change these values in the eclipse.ini file:

```
-Xmx2048m
```

1.4 Tracing a program

It is suggested to use lttng-simple to trace a program. The script runs the supplied command under tracing. In consequence, the tracing is active only for the duration of the program to study. The trace is created by default in the current directory, and is inside a directory with the name of the program and a suffix to indicate « -k » for kernel, « -u » for user-space and « -ku » for both. An event list can be supplied to select the desired events. Here is an example :

```
# will create the trace sleep-k/ with default events and system calls
lttng-simple -k -s -- sleep 1
```

2 System calls

2.1 Microcontroller

An embedded, micro-controller, such as Arduino board (fig. 1), has only one main loop program. It has exclusive access to all resources, including memory and I/O. This simple design needs only small amount of memory, but has several limitations.

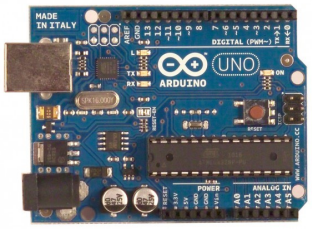


Fig. 1: Arduino board with Atmel microcontroller

Situation	Embedded board	x86 + Linux
Can a program access memory of other programs?		
What happens if a program never terminates?		
What happens if a program crashes?		

2.2 System calls

The system call interface is the actual API of the operating system. They allow to trigger kernel code routine to obtain resources such as file, memory and devices. It needs the support of the hardware to operate and enforce the security boundary.

A system call is a type of **software interrupt**. When a system call is invoked, the processor jumps to a function that calls the requested system call based on its number. When the processor switch to kernel mode, the memory that belong to the kernel becomes accessible. When the system call is finished, the return value is written into a register, and the program resumes its operation.

2.2.1 Activity 00-minisys : Raw system call in assembly

Usually, system calls are wrapped in the C standard library. But how does this library is actually working? To experiment this, we use the program `minisys.asm` that does raw system call in assembly. The calling convention for the current architecture must be used. Here is an example of the write system call, that outputs a string on the standard output.

```
    mov     rdi, stdout         ; argument 1 : file descriptor
    mov     rsi, msg            ; argument 2 : string pointer
    mov     rdx, msg_len        ; argument 3 : string length (bytes)
    mov     rax, sys_write      ; system call number
    syscall                     ; toggle to kernel mode
```

Notice the special instruction `syscall`, which is specific to the architecture x86-64. In previous Intel assembly, the equivalent instruction was `int 0x80`. The exact system call number of `sys_write` depends on the kernel itself. When booting, the kernel programs the system call vector, an array of function pointers. The value in register `rax` actually specifies the index in the array to use for the jump. The following file in the Linux source tree contains the system call numbers for the x86-64 architecture :

linux/arch/x86/entry/syscalls/syscall_64.tbl

This list indicates that `sys_write` has the number 1. This sytem call takes three arguments, passed in registers. There are at most 6 arguments in registers. The return value is in `rax`.

Once the processor executes the syscall instruction, then the following code is executed, from the file linux/fs/read_write.c :

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
{
        struct fd f = fdget(fd);
        ssize_t ret = -EBADF;

        if (f.file) {
```

```
                loff_t pos = file_pos_read(f.file);
                ret = vfs_write(f.file, buf, count, &pos);
                file_pos_write(f.file, pos);
                fdput(f);
        }

        return ret;
}
```

Exercices :

- Implement the call to the raw sys_nanosleep system call inside minisys.asm. Specify a short sleep duration, such as 10ns.
- What happens if sys_exit is commented? Restore the system call after the experiment.

2.2.2 Tracing system calls with strace

The strace tool displays system calls made by a program. The arguments are decoded in strings and labels, for convenience. Here is an example with `minisys`.

```
# redirect program output to /dev/null
strace ./src/minisys > /dev/null
```

The strace command takes the program to execute as a parameter.

Exercices :

- What are the system calls made by minisys?
- What is the return value if an invalid file descriptor (like 42) is passed to sys_write? Basic error codes are in the file linux/include/uapi/asm-generic/errno-base.h
- The C program `hello` does essentially the same thing as `minisys`. Compare system calls made by the two programs.

2.2.3 System call tracing with LTTng

LTTng records events occuring in the operating system, including system calls. Unlike strace, that rely on ptrace mechanism wich stops the program each time a system call occur, kernel tracing does not stops the monitored application.

Tracing minisys is simple:

```
$ lttng-simple -c -k -s -- ./src/minisys
```

The trace is recorded into the current directory and has the name `minisys-k`. This trace can be imported into TraceCompass. For the details, refer to the *LTTng Plug-in User Guide* in the TraceCompass manual.

Exercices :

- Locate the minisys process in the Control Flow View, and look at the system calls.
- How much time takes the write system call?
- What is the duration of the nanosleep system call? What is the relative error between the actual delay and the requested delay? Explain the difference.
- Redo the experiment, now using Python. What is the underlying system call to `time.sleep()`?

3 Processes and threads

How does new programs are started? What is the difference between processes, threads and user-space threads? What is the effect on the memory space and performance? This activity aims at answering these questions.

3.1 Activity 01-banque: Bank operations

The program banque simulates transactions on a bank account shared by multiple ATMs. Four modes are available: serial, processes (fork), threads (pthread) and user-space threads (pth). The serial mode, where each ATM is making their operations at a time, is implemented. Your goal is to implement the other modes. Follow instruction in the code. Trace the resulting program:

```
lttng-simple -k -c -s -- ./banque-all.sh
```

Then, fill the following table. Explain the observed behavior.

Mode	Number of PIDs created	Number of TIDs created	Is execution parallel?	Is result correct?
serial				
fork				
pthread				
pth				

3.1.1 Measuring cache misses

The script banque-perf.sh reports the L1 cache misses for each ATM mode. Is there a large difference between modes? Explain the results obtained. Hint: this hardware performance counter is accessible only from the host, but not inside a virtual machine.

3.1.2 System calls involved in creating tasks

What is the difference in terms of system calls between creating a process and a thread? While the kernel trace has the data, strace decodes system call arguments, therefore we suggest to use strace to answer this question. Copy-paste the result below. You can check the meaning of the flags from the man page.

3.2 Activity 02-chaine: Starting new programs

Your challenge is to execute three different programs, one after the other, without creating new processes for each of them. The programs are foo, bar and baz. Each program uses whoami() function to display information about the task, including the PID and the value of the global variable rank, incremented by each program. The master program is chaine, but currently, it only executes the program foo. Fix it to execute the other programs in sequence.

Check using a kernel trace the result of the program's execution.

```
lttng-simple -k -s -- ./chaine
```

The variable rank is incremented in each programs. Then, why the value is always the same?

4 Locking and synchronization

4.1 Activity 03-multilock: classification of locks

You are working on a statistic library, where multiple threads can add samples at the same time. Therefore, it is necessary to protect the critical section using a lock. In this activity, we study the difference between a mutex, a fair semaphore and a minimal spinlock implementation. The spinlock implementation is provided, but you are invited to look at the assembly. Follow instructions in the files mutex.c, semrelay.c and spinlock.c. Each file has an init and done functions, where the lock must be created and destroyed. The worker function implements the main loop that access the locks. There are two loops. The outer loop repeats the critical section and the inner loop simulates work inside the critical sections. You can check your result with the “--check” option. Then, trace the program, and interpret the data in TraceCompass. There are two executions with different outer and inner parameter.

```
lttng-simple -k -s -- ./multilock-all.sh
```

Property	Mutex	Semrelay	Spinlock
Does blocking visible? (WAIT_BLOCKED)			
Is there system calls inside the critical section?			
Is there visible starvation?			

4.2 Activity 04-interblocage: deadlock

Deadlock occurs when a wait cycle is created. In this activity, we create a deadlock between two threads and two mutexes, by changing the order mutexes are locked. The deadlock may not occur if one thread completes its critical section before the other:

Thread 1	Thread 2
lock(a)	
lock(b)	
unlock(a) unlock(b)	
	lock(b)
	lock(a) // no problem

However, if the locking is interleaved, then a deadlock may occur:

Thread 1	Thread 2
lock(a)	
	lock(b)
lock(b) // already locked by thread 2	
	lock(a) // already locked by thread 1, deadlock

The goal is to detect when the deadlock occurs, and then quit the program. The detection is made into a timer signal, because even though the thread is blocked, signals can be delivered to the program. Inside the watchdog() signal handler, implement the detection algorithm.

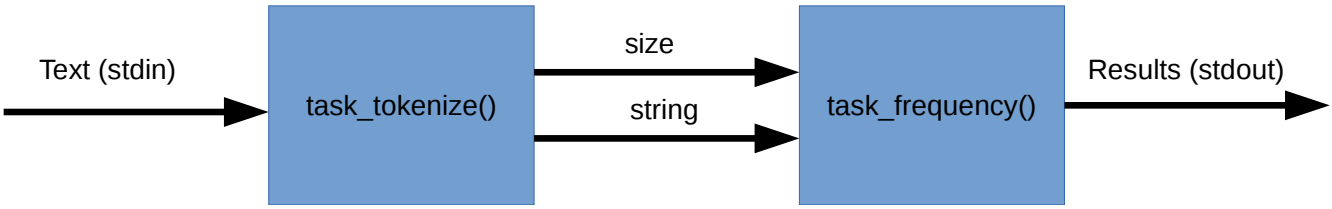
How to test the deadlock algorithm? Because the deadlock may never occur, the test may never terminate. In order to test the detection algorithm, force the deadlock using a barrier, such as pthread_barrier_wait().

4.3 Activity 05-lexique: Producer and consumer over pipes

Lexical analysis is the first step for classifying documents. For instance, word frequency can be used to

detect the topic of a text, and find similar ones.

In this activity, the lexical analysis is done by two processes, communicating by pipes, as shown in the following figure. The initial text is received on the standard input by task_tokenize(). The first step is tokenize the input into words. The size (length in byte) of the word is written in a pipe, and the word itself is written in a second pipe. The task_frequency() reads the words and put them into a hash map for counting them.



The challenge is to start the processes and connect them with pipes. Take care of the convention of the binary format used to communicate the size and the string. Finally, if a user hits CTRL-C, the program and its children quits without displaying accumulated results. Implement the SIGINT signal handler to quit gracefully. Here is an example of program execution:

```
./lexique < text-small
```

Questions:

- Trace the resulting program to evaluate its run-time behaviour. Would you consider this architecture as efficient?
- Let's consider that task_tokenize() is written in C and task_frequency() in Java, on Intel x86-64 architecture. What is important to consider when communicating the data?
- Determine experimentally the size of the pipe buffer using the program remplissage.c.

5 Virtual memory

The virtual memory is an indirection level from the actual physical memory. The memory is real, the virtual term refers to the address translation that occurs. The Memory Management Unit (MMU) translates virtual addresses to physical (bus) addresses in hardware, using the page table as the reference for the mapping. The operating system managed the page table. On Intel x86-64, the CR3 register indicates the location of the page table, as shown in the following picture. Actually, from the point of view of the processor, a process switch consists essentially in changing the CR3 register. Multiple threads share the same page table, while different processes have different page table.

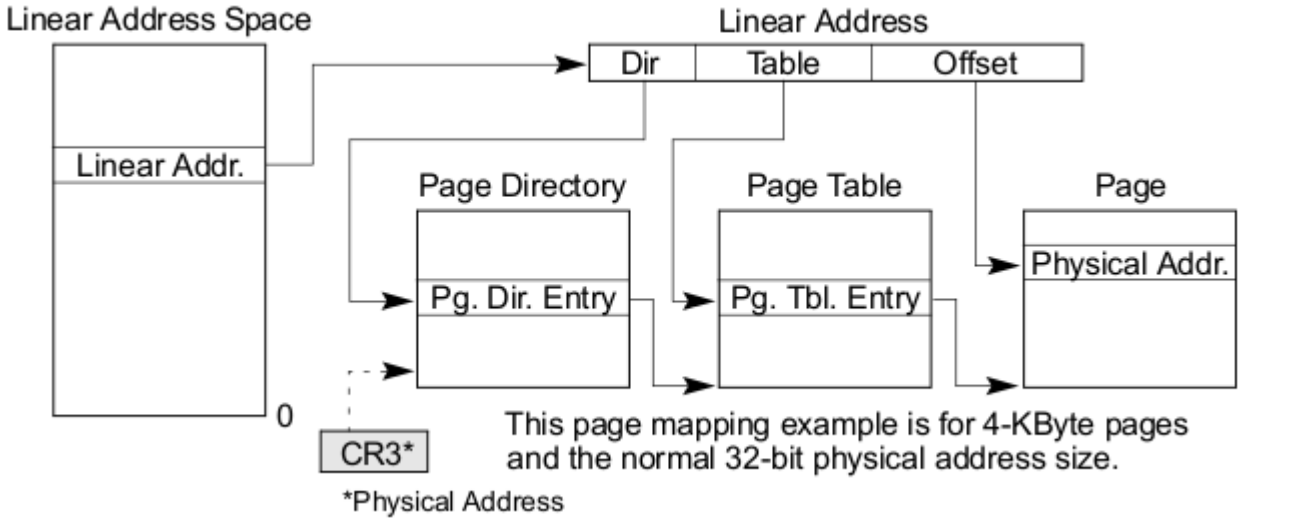


Illustration 2: Intel x86 page table

(Source : Intel® 64 and IA-32 Architectures Software Developer's Manual)

As long as the process access memory that has a mapping to an present page in physical memory, the operating system is not involved. When the process tries to access a memory that is outside the valid range, a page fault occurs (also called trap), which is a type of software interrupt. The handler can either:

- The address is invalid: send SIGSEGV to the program (dereferencing invalid pointer)
- The address is valid, but the physical page is not allocated: allocate a page, update the page table and resume executions (minor page fault)
- The address is valid, but the physical page is not present in main memory: perform paging from block device, which usually block while the data is ready (major page fault)

The page fault must not be confused with cache miss. The former is a concept related to the operating system, while the later is a micro-architecture level event that causes the pipeline to stall.

5.1 Activity 06-pagedump: dumping memory

In this activity, we dump entire page content to see how the layout of allocation in memory. Complete the small utility pagedump.c to write to disk the content of the page of a given address. The memory content is set to distinguishable values that are easy to find using an hexadecimal editor, such as okteta or hexdump.

Questions:

- What are the differences between the variables on the stack and the heap?

  Stack variables are defined contingously and in the begginning of the execution while heap variables are not defined contigously.

- In which cases the data is contiguous?

  Stack: contigous, heap: not contigous

- Does a thousand malloc(1) is equivalent to one malloc(1000)?

  Not at all, because malloc produces the alocation at the heap, thefore it is not the same.

5.2 Activity 07-drmem: memory demand and allocation

How does the operating system respond to the memory demand of a program? To observe the behaviour, in this activity we use a special program called drmem, that simulates various memory demand, on the stack and on the heap. The program is traced by using three different tracepoint:

- The real memory allocated to the program is measured using kernel page allocation events (kmem_mm_page_alloc and kmem_mm_page_free). Each event indicates that one page is added or removed from the page table of the process.

- The memory allocated on the heap is traced using LTTng-UST libc wrapper. It works by preloading a shared library, that overload the malloc() and free() functions.

- The memory on the stack is traced using LTTng-UST. On each function entry, the tracepoint records the value of the stack pointer register (rsp). The actual hook on each function rely on the -finstrument-functions feature (see man gcc).

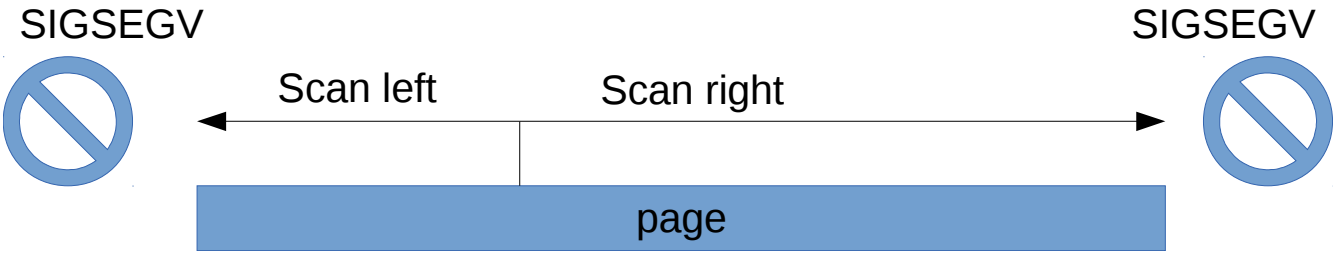
Before running experiments, make sure Python binding for Babeltrace are installed:

```
sudo apt-get install python3-babeltrace
```

The memory experiments are in the file experiment.py. This script manages the tracing session and output PNG charts under the results directory. Refer to the experiment examples in this script and drmem.c for the parameters (location (heap or stack), allocation size (byte, word, page, huge), trim and fill). The delay parameter is to slow down the execution, in case you have lost events, and make nicer charts. Your goal is to make various experiments to explore the effect of each parameters, then explain the observed behaviour.

5.3 Activity 08-segfault: crashing for fun

The signal SIGSEGV is delivered when the address accessed is invalid. This activity aims at understanding exactly what type of invalid access triggers this signal. The utility segfault scans memory from a given, valid location, until the SIGSEGV occurs. Inside the signal handler, your goal is to show the memory address that caused the segfault, and its offset from the provided address. The maps file is dumped in a file on each execution to understand where the failure occurs.



Question:

- Could it be possible to use the address zero (NULL)?
- Let a program has an array int buf[10], where indexes from 0 to 9 are valid. Does accessing buf[10] will automatically cause a segmentation fault?

5.4 Activity 09-randaddr: address randomization

For security, Linux randomizes the address space, such that on each execution, the address space is different. It complicates the security attacks based on fixed address. However, to study such attacks and for debugging programs, we would like to prevent address randomization. Create a shim that spawn the command passed as parameter without address randomization. Achieve this with the personality() system call.

Adresse type	Randomized?
Global variable	
Stack variable	
Heap variable	
Code in .text section (main)	
Code in share library (a, b, c)	

Questions:

- The dummy program displays symbol addresses of major memory locations. Which one are

affected by address randomization?