Embedded Linux

Techniques

Hands on course for the Linux c developer

10/8/2012

Raz Ben Jehuda

raziebe @ gmail.com

TOC

1. Introduction

This book is a guide for the newcomer in Linux. It provides guidelines and heuristics how to program in Linux.

Usually when we refer to embedded devices we imagine a device dedicated to perform a specific task. But in the past decades the requirements from embedded systems had grown from a simple command line interface to a touch screen running Android and now virtual reality glasses and Robots.

Therefore, many engineers that move to Linux from another operating system find themselves busy in tasks they'd hardly ever done before. Things like configuration, scripting, PC hardware awareness, system debugging, open source assimilation, real time constraints, performance and scalability become much more critical than before.

The first rule is – use a Linux workstation. If you must use windows install Linux in a virtual machine.

The course can be cloned from:

git://git.code.sf.net/p/linuxdebug/code

2. Open source

Linux distributions are known to use an open source software. Many products are shipped with some open source packages. The common steps for using a package is to download it, compile it, test it, and then integrate it into your executable.

2.1 Get the software

For many programmers downloading a code usually means clicking on the link of the tar ball or zip file. There are times though that it is a bit more complicated. For instance, when the computer used to build the software has no GUI but only accessible through a terminal, like putty or Cygwin in windows, or through a secure shell in Linux. Obviously, It is possible to download to the PC and then to the build machine, but another way is to use the command “wget” in the shell.

Nonetheless there would be occasions when a certain branch of the code is needed and not the main trunk. Many open source projects today are handled by git, so it is quite often to get sources with git.

2.2 Documentation

Next step would be to read the documentation. Many projects are shipped with a documentation library or a README file. Read it. It might save you a lot of trouble, because there are times where the author will list some constraints and advices. Another important issue is the software license. For example, there is a big difference between GPL and BSD licenses.

2.3 Build

If possible build first in your Linux PC if the software is runnable in x86 or x86-64. This saves a lot of time, and the PC can be used as a reference machine (if possible).

Two main build suites are Automake and cmake. Both suites would first ask you to configure the software. Configuration usually mean enabling or disabling certain features, choosing the compiler etc.

Once the configuration is done run make. Do not install the software in your PC without directing it to an empty directory as it might collide with a current software.

2.4 Experience

Execute the runnable. It can be an executable or as shell script that wraps an executable. Run it according to the documentation. Now experience the software and check if it answers all your requirements. Test it.

2.5 Integrate

Third party software integration needs to be done with caution. If for example, the software has a GPL license and you need this software to access internal variables in your executable you need to create IPC calls to software or expose your variable through shared memory. A common example here is having a web server configure the executable.

The package needs to re-configured to suit the target. You may choose to disable some features and change the compiler.

Download curl. Compile it for raspberry PI or qemu.

How is the software licensed?

3. Shell commands

Embedded Linux programmers tend to wrap their binaries in some launching scripts. Thus, at some point a programmer is likely to find himself reading scripts.

This is a short overview for shell scripting you will find in most Linux machines but it is definitely does not cover shell scripting. This short summary is for bash scripting.

3.1 IO redirection

IO redirection is performed by using the operators: <,>,| , >> .

The sign ">" is used to direct the output of a command to file. For example:

```
$ ls > /tmp/lsout
```

The sign < is used to direct the input of a command from some other source.

```
$ grep root < /etc/passwd
```

It is also possible to combine redirection in a single line.

```
$ sort < /etc/passwd > /tmp/output
```

Errors filtering

the bellow command will generate a lot of errors:

```
$ find /proc/
```

To remove all “Permission denied” errors we redirect the standard output to /dev/null:

```
$ find /proc/ 2>/dev/null
```

Commands chaining

There are times that we want chain commands, for this we use the sign “|”. For example:

```
$ cat /proc/interrupts | sort –k 3 | tail –n 10
```

We can also choose to redirect output to the standard input or to the standard error. And to illustrate:

```
$ ls 2>&1
```

3.2 Operators

It is expected that each shell command returns 0 on success else it returns an error value. To get the error value of a command we print:

```
$ echo $?
```

There are cases of commands chaining where we wish a command to be invoked only if the previous command was successful. In the case we use the operator “&&”, but in this case 0 (success) means true. For example:

```
$ ls thefile 1>/dev/null && grep me < thefile >/dev/null && echo found it
```

Another operator is ||. This operator would invoke one command a time until the first success.

3.3 Scripting

Shell script are usually small text files starting with the prefix.

```
#!/bin/sh
```

This prefix instructs the command line interpreter to run this text file as shell script "sh". For instance, the command: $ ./myscript.sh would run a text file as a command script but the script file must have its execution bit set.

To debug a shell script a programmer can run the shell with –x argument like that:

```
$ sh –x ./myscript.sh
```

To write shell script you should use vi, pico or nano editors. It is not customary to write shell scripts in eclipse-like editors because programmers tend to fix the scripts in the target machine.

3.4 Passing arguments to a script

Arguments are referred by their index when $1 is the first argument. The number of argument is symbolled by $#.

3.5 Variables

Variables need not to be pre-declared. Assigning a value to a variable is done with the operator "=" and referred to by prefixing them with the $ sign.

```
rc="hello"

echo $rc
```

3.6 Conditionals

The syntax of if-then-else is as follows:

```
if [ "$rc" = "1" ]; then
	echo is one
else
	echo not one
fi
```

Make sure to end the right bracket "];" .

3.7 Loops

There are two loops syntaxes. Do-while and for.

```
for i in $(cat myfile); do
	echo word $i
end
```

A while loop should look like that:

```
c=0
while [  $c -lt 10 ]; do
        echo c is $c
        c=$((c+1))
 done
```

3.8 Functions

Functions syntax is as follows:

```
func() {
}
```

To call it we simply write its name: func

3.8.1 Local variables

The bellow example passes a single argument to func and launches it in the background.

```
func() {
        local x=$1
        echo FUNC $1
}

func arg1 &
```

Wrap the following line in a function in a launching script. What does it do?

```
$ mkfifo /dev/mypipe;
```

```
$ cat /dev/mypipe | sh -i  2>&1  |  netcat  -l 10001   > /dev/mypipe
```

4. Processes

A process is a program in the state of execution. There can be several processes instances of the same program. Processes contain various kind of resources, such as address space, descriptors, pending signals and internal kernel data. Every process has one or more threads that execute it.

As noted a program binary code may be shared between many instances of processes. To examine a process memory, enter:

```
$ cat /proc/<some task pid>/maps
```

You will notice there some libraries mapped to addresses. Some of these libraries are shared between processes.

What is actually shared in a shared library?

Write a program and a shared library with some global variable, and static variable. Access and change this variable through the program.

Are there any changes visible between processes?

Process states

A process state machine can be simplified:

RUNNING ←→ (NON) INTERRUTIBLE ←ZOMBIE

A Process can also be in a STOPPED mode.

4.1 Process Creation

A Process invocation is performed by two system calls. The clone system-call and then exec that loads a new program code into the new address space.

Another system call is fork. Linux implements fork by using the copy-to-write (COW) technique. Upon "fork" all of the parent process resources are duplicated and the copy is delivered to the child process. This approach is inefficient because if the process loads a new binary image then all the copied data will be thrown, and if the child process can share some data with its parent then the copy memory is a waste. COW delays the copying of the data only when it is written to. This way only dirty memory of entirely new data is allocated.

4.2 Process Termination

As all living creatures, all processes must die. A process might end its life voluntary by calling exit, or in a more violent manner – being killed by some signal or an exception.

When a process terminates all of its resources are begin released. A Resource such as a memory; timers, semaphores, futexes, opened files, are dereferenced. Signal handlers are being released and any other object is dereferenced.

Once a process sets its exit code and signals its parent it becomes a zombie and then calls schedule. The zombie process exists as long as its parent did not call the wait4 to collect its exit code.

Hamm…

But what happens if the parent dies before its children?

5. System calls

A system call is a request made a process from the kernel. There are two main types of system calls in Linux, a real system-call and a virtual system call.

5.1 Real System call

A simple system-call in x86 is performed by calling the assembler command INT 0x80. Here is an assembler code that performs that call fork system call.

```
 #include <stdio.h>

int main()
{
      asm(        "movl $2,%eax;\n"  \
                  "xorl %ebx,%ebx; \n" \
                  "int  $0x80\n"
          );
      printf("%d \n",getpid());
}
```

System call flow is as follows:

1. Assign all input structures onto registers according to processor architecture.
2. Call the software interrupt trigger assembler command
3. Collect the return value of the system in the register according to the architecture.

Implement the close(fd) system call as above. Make sure the system call succeeds by checking the return value.

5.2 Virtual system call

A second type of system call is called virtual system call, or vDSO. A virtual system call does not reach the kernel but it is actually a code copied into the process address space and executed as a function. There is no software interrupt. The reasoning here was that in some machines system calls cost too much time. Also; there are certain kernel configurations in which when a system call is made the process may lose the processor.

You are introduced to new Linux machine and you wish to know if it supports vdso or vsyscall.

How can you check it?

Gettimeofday is commonly used for profiling. For this reason Linux implements gettimeofday as a virtual system call.

Prove that.

6. Threads

6.1 Threads basics

To create a thread, we use pthread_create. To stop a thread we use pthread_stop. Pthread_t is the token in use in the Posix APIs to manage threads. It has no relation to any kernel resource.

A thread id is the return value of “int gettid(void)”. This is can be viewed by issuing the command:

```
# ps –efH
build   9208  9207  0 00:00:00   ? 07יול   (sd-pam)
root    9682     1  0 00:01:46   ? 07יול  apache2 -f /opt/git/SDK.git/gitweb/apac
root    9683  9682  0 00:10:47   ? 07יול   apache2 -f /opt/git/SDK.git/gitweb/ap
root    9684  9682  0 00:10:45   ? 07יול   apache2 -f /opt/git/SDK.git/gitweb/ap
root    9685  9682  0 00:10:42   ? 07יול   apache2 -f /opt/git/SDK.git/gitweb/ap
root    9777  9682  0 00:10:54   ? 07יול   apache2 -f /opt/git/SDK.git/gitweb/ap
```

There are cases where gettid() does not exist. For this use syscall wrapper defined in libc.

Thread stopping is not thread killing. Linux does not support a killing of a thread. This is because a thread has resources allocated (mainly its stack) which will leak. For this reason, when a thread needs to be killed we should implement exits points.

6.2 Scheduling

The scheduler is an algorithm that selects which task to execute next. The scheduler main purpose is to give the illusion that many processes execute concurrently. In a single processor computer only a single process can run at a time. In a multicore or a multi-processors machine two or more distinct processes can execute concurrently. Linux implements a scheduling conception known as Preemptive multitasking. Preemption means that it is up to the operating system to decide which process to run next. Scheduling algorithms have a policy, this policy determines which process should run next.

What is so important in preemption? !

Processes can be classified as I/O bound or as cpu bound. A policy of a scheduling algorithm for example, may favor I/O bound processes (which is usually the case).

Can you think of a way to show that Linux tend to favor I/O processes over processor bound processes? !

6.2.1 Process Priority

A common type of scheduling is priority based. Higher priority means longer times slices in addition to favoring the highest rank process though not necessarily consequent. When two processes have the same priority, they are scheduled one after the other in a round robin manner. Linux scheduler implements a dynamic priority and static priority. Dynamic priorities are auto-calculated in real time. For example, I/O bound processes are boosted by the processor. Static priorities range from +19 to -20 when -20 is the highest priority. These priorities are manipulated by the nice* command.

The scheduler also implements a load balancing algorithm when it comes to multicore computers. The Load balancer moves one or processes from one core to another. Each core has its own run queues and a process may be moved from processor to another.

Hamm..What do you think might go wrong with load balancing?

There are cases where a programmer chooses to bind a process to a processor. Binding a thread to a processor is done by a Posix APIs or by the taskset command.

In the real-time world programmers leave a trap door in a form of a thread/process with the highest priority possible? Why does this process have the highest priority?

6.2.2 Real Time

There are two scheduling policies: round robin and a fifo. In a round-robin a task runs until it consumes all of its time period while a fifo a task would run until it yields the processor. When a task consumes its time slice or gives up the processor a next task with the same real-time priority or higher is the candidate to run. A low priority task may not preempt a round robin real time task even if its time slice is consumed.

Write a program that performs a busy loop in a fifo priority and run it on a single processor machine. What happened? Why?

6.3 Thread names

Naming threads is important because many times we will need to control their priorities and other attributes from the command line. To name a thread we use the prctl API. To change a thread's priority, we use chrt.

6.4 Atomic operations

Atomic operations like an increment or a decrement are implemented by by gcc built-ins functions. For example, to increment:

```
long val;

__sync_fetch_and_add(&val, 1);
```

In cases when a certain function decrement or increment is not implemented the compiler would generate a warning and a call to an external function would be generated. The function name would be same with an additional _<size of type> at its end.

Write a program that increments an integer up to the value of 1000000:

```
function addVal ( long *val, long *add){
*val ← *val + *add;
}
```

First do it by using simple addition, then __sync_fetch_and_add and then do it by wrapping the summation code by a mutex. Use a single thread, not two.

You need to average all increment, save the minimum and the maximum.

What is the worst case?

6.5 Semaphores

In Linux, there are two types of semaphores, system V and Posix semaphores.

6.5.1 Posix Semaphores

Posix semaphores are counting semaphores. These semaphores are described as light weight semaphores because they are implemented in libc and as long as there is no need to relinquish the processor they do not do any system call. Posix semaphores may be shared between processes in two forms, use names or use a shared memory.

6.5.2 System V Semaphores

System V (pronounced system five) were widely used few decades ago in Unix OS systems. Today they are considered obsolete.

The way to use these semaphores is by creating a named semaphore or attaching to it. Then perform decrement or increment through a libc APIs system call wrappers. The naming makes it possible to share the semaphore between processes.

Every function call is actually a system call and because of that Sys V semaphores are considered resources consumers.

One difference between Posix semaphore and system V semaphores is how the thundering herd problem is handled.

Read about this problem and implement a test code.

What is the difference?

6.6 Inter Process Communication

There are times where we need to share information between processes. There are several methods to do that.

6.6.1 System V IPC

System V shared memory was widely used few decades ago however today it is considered quite obsolete. Nonetheless Sys V IPC is very efficient and easy to use. In sys V a shared memory is created with a given key and then it is attached by any other thread using this key. One disadvantage is the fact that this memory is not a file and as such it cannot be accessed by command like “cat”. However; a big advantage is that there are little authentication restrictions.

6.6.2 Mmap a file

Another way to share memory is by mapping a file to a processes address space. A file is created or opened and a mmap system call is made on its descriptor. It is important to have the file over tmpfs file system because this way you can be sure no writes to the disk are made. Accessing the file is done by referencing a pointer in the code. A big advantage is the fact it is possible to examine the shared memory by using the “cat” command that simply prints the file contents.

It is notable to say that tmpfs is a RAM-only file system which is not preserved after the file system is being unmounted - it is does not survive a computer restart.

6.7 Messaging

There are several ways to pass data between tasks in Linux. Each method has disadvantages and advantages. It is important to make the correct decision of which messaging method you wish to use. Some may support priorities, some are multiusers and so on.

6.7.1 Fifos

Fifos are distinguished from other message passing methods by the fact that there can only two users of a pipe, a writer and a reader. Each pipe descriptor has a direction.

One user opens the pipe for reading and another opens it for reading.

```
tty0$ mkfifo mypipe
tty0$ cat mypipe
tty1$ echo Hello > mypipe
tty0$ Hello
```

6.7.2 Pipes

The pipe function returns an unnamed pipe, which is represented by two file descriptors, one for each direction. This sort of a pipe is useful when a parent process needs to communicate with its child process. The child program usually is a c code forked and its descriptors inherited.

The popen function is another way to pass messages between a process and its forked child. Popen service function returns two descriptors one for reading and the other for writing. Popen is useful for sending shell command and grabbing the shell's output.

6.7.3 Sys V messaging

Sys V messages were widely used few decades ago, however today they are considered obsolete. It use is as follows: A user defines a message queue by the msgget system call with some identifier, then the process can send or get messages.

6.7.4 Posix Message queues

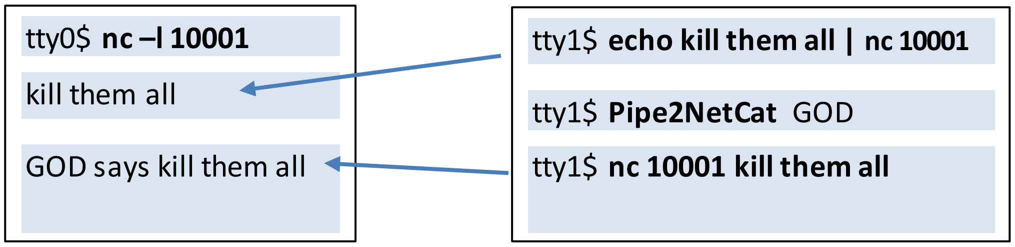
Message queues are a mean to pass information between two or more tasks. A user calls mq_open with a certain key, gets a special descriptor and uses it to pass messages.

6.7.5 UNIX domain sockets

Unix domain sockets are from socket type family AF_UNIX. Their main use is to be a medium of communication between two or tasks on the same machine. A Unix domain socket can be named or unnamed. They are considered light weight.

Multiplexing descriptors

Netcat is a program that is used to pass textual data to another program over ip (tcp or udp).
Modify netcat.c code to support a dynamic prefix. The prefix must be provided by a fifo or a pipe or a socket to netcat.



```
tty0$ nc –l 10001
kill them all
GOD says kill them all
```

```
tty1$ echo kill them all | nc 10001
tty1$ Pipe2NetCat  GOD
tty1$ nc 10001 kill them all
```

You need to modify the readwrite routine in netcat.c to be able to poll on one additional descriptor.

6.8 Mutual Exclusion

Many times, it is necessary synchronize access to a shared resource between threads. We do that by creating mutexes. Linux support Posix mutexes. Mutexes have attributes. Mutex can be recursive, spinning, shared between processes and robust.

A recursive mutex is a mutex that supports multiple locking by the same owner. Hamm... There are programmers that consider this a bad practice. Why?

A known dilemma is what is the amount of code that we surround with protect in locks. The rule of thumb is to reduce the amount of work performed by the system as little as possible.

A spinning mutex is a mutex that spins instead of giving up the processor.

Who needs that ? What are the risks ?

For a mutex to be **shared** between processes its descriptor pthread_mutex_t must be created in a shared memory. Why?

A **Robustness** of a mutex means that if a mutex owner dies while holding the lock the operating system unlocks the mutex and wakes up a single waiting thread with the appropriate error code.

6.8.1 Priority Inversion

Priority inversion is a known problem in computers sciences. Priority inversion happens when a high priority task is preempted by a low priority task. Here is the scenario:

There are three tasks A,B and C. when A has the highest priority and C has the lowest priority.

1. C grabs a lock. C runs and then A is scheduled.
2. A runs and then tries to grab C's lock. A gives up the processor.
3. C starts to run but before it releases the processor it is preempted by B.
4. B preempts A.

There many solutions to this problem. Here are some:

1. <u>Disabling interrupts</u> all together. By disabling interrupts, we have only two priorities. This solution is not practical in multiple processors machines as well as disabling interrupts in user space is highly discouraged.
2. <u>Priority ceiling</u>. This approach gives the lock object itself a priority. Any task that grab the lock gets this priority. It is assumed that all tasks that use the lock have a lower priority than the lock.
3. <u>Priority inheritance</u>. The lock owner is assigned temporarily the highest priority of the highest task waiting. This solution is implemented in Linux.

4. Random boosting. Lock owners have their priorities randomly boosted.

Priority inversion was notoriously known as the error that caused the malfunction in “Mars Pathfinder”. A watchdog timer booted the entire system.

Create a simulation to this problem and then fix it.

6.9 Multicore Programming

We already saw that threads and processes can be assigned to certain processors. It is considered a good programming to make your program aware to the computer's processors. This can be achieved in compile time or in real time or both.

To find out how many processors a computer has we can count the number of columns in /proc/interrupts. Another way is to use sysconf command.

Write a simple c program that reads the number of processors using sysconf.

Make your program cpu-cache size aware in compile time by using the bellow:

```
getconf LEVEL1_DCACHE_LINESIZE)
```

A nice visualization tools is lstopo.

7. Signals

A signal is a mean in Linux to pass events to a process from another process or from the kernel.

There are two types of signals, regular signals (or unix signals) and real time signals (Posix signals). Real time signals are queued while regular signals don't.

To list the supported regular signals in enter:

```
$ kill -l
```

There are 32 regular signals (1 to 31) and 32 (indexes 32 to 64) real time signals.

A regular fault that is generated by the processor usually reflects sort of an error in the running program. Like a segmentation fault or a division by zero. These faults are caught by the kernel and passed to the process through the kernel exception handler. Software generated signals are signals sent by a user by the kill or tgkill (for a thread group) system calls.

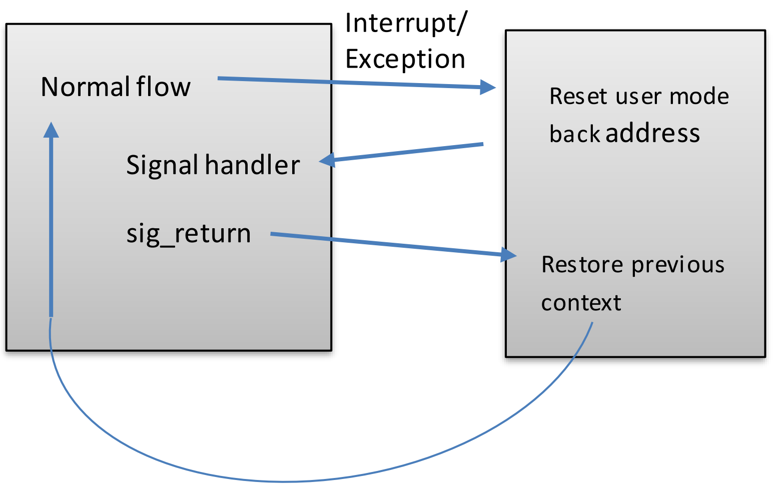
Regular signals of the same kind are never queued and overrun one another. Signals may be blocked, ignored, polled or asynchronically handled by a signal handler routine, kill the target process, stop it or resume it.

When a process is not executing and a signal is sent to this process the signal is a pending signal – even if it is blocked the kernel would register it in the task information descriptor. Also, while the signal is delivered and handled by the signal handler it is being blocked.

A signal is considered a **private signal** if it is sent to a task (thread) and **shared signal** it is sent to a thread group (aka the process).

7.1 Signal Delivery

A signal is delivered to a process only when the processor returns to user mode from an interrupt or an exception. If there's a signal, the kernel would replace the current user stack with the signal handler stack (the return address of the process was modified).



As you can see, signals are complicated and consumes lots of resources, this is why we should use them carefully.

7.2 Returning from system calls

When a process issues a system call the system call may not return immediately and the issuing task may be blocked. If the process receives some signal it will finish the system call and returns EINTR.

Programmers reading the man page of signal or sigcation are a bit puzzled in regard to which system calls and under what circumstances does a system call returns disturbed by a signal.

Please write a small piece code.

1. Define signal handler with sigaction
2. By using sigcation set sa_flags to SA_RESTART
3. rc ←sleep(100)

$ kill –SIGTERM <task pid>

After that replace sleep with a different system call.

8. Linux RT Preempt

Real time system is one in which the correctness of the computation does not depend entirely on the logical correctness of the computation but also upon the time at which the result is produced. Linux RT-Preempt provides the real time accuracy up to a few microseconds with a suitable hardware. It is important to note that real time does not mean fast or high performance. Real time only means accuracy and predictability.

8.1 Real time measures

The following sections describe the how a real-time operating system is measured.

8.1.1 Latency

Latency is the amount time that passes between the event and the moment it is being processed. In most cases events are hardware interrupts. An interrupt latency is the amount of time passes between the hardware event to the software handler to process it. In addition; there is also a dispatch latency which is the amount of time it takes for the handling task to launch once the interrupt handler finished. A non-real time Linux will not necessarily launch the task that needs to process the event once the interrupt is completed; Only when the current process reaches a rescheduling point it will dismiss the processor.

8.1.2 Preemption granularity

Preemption granularity is the amount of time it takes for the kernel to move from a low priority task to a high priority task. In the early days of Linux the kernel could not just jump from one task to another and the context switch took place only upon system call exit. This kernel is called a **non preemptive kernel**. Later **rescheduling points** were added. Rescheduling points are common primitives (like spin unlock) in the kernel that whenever used they also check whether a task should be switched or not. This approach was known as preempt-able kernel. Preempt-able kernel major flaw was that if a low priority task holds some lock exclusively this task cannot be preempted. Preempt-able kernel offers good average but not real time. Linux Preempt RT solves this problem. In Linux Preempt RT there is no global lock (no preemption disabling) and all tasks are independent one from the other.

8.1.3 Interrupt disabling

As we saw previously interrupt latency is the most important measure of the efficiency of a real-time system. A hardware device will not generate an interrupt before the current interrupt is handled. A known problem is that the kernel code tends to mask interrupts when entering a critical section. Thus, interrupts are being delayed on a running processor quite frequently.

8.2 Preempt RT achievement

Linux preempt RT addresses all of the above problems. It does so by:

1. All execution contexts were replaced by preempt-able threads.
   a. Softirq
   b. Tasklets
   c. IRQS - hardware interrupts requests. Only mask the interrupt and wake the handling thread.

Once all are execution contexts were preempt-able there is no need to mask interrupts and it actually possible to assign priority to an IRQ. IRQ Latency is reduced because once the IRQ returns the kernel would check which thread should run according to its priority.

Code & run the following code:

```
do {
        t1 ← gettime
        sleep 1ms
        t2←gettime
while (t2-t1 < 1200us )
```

Execute this code in native kernel and in RT Preempt Kernel.

8.3 From VxWorks to Linux

This section is intended for VxWorks programmers that wish to port their product to Preempt RT Linux. It is a bullet list of various possible caveats.

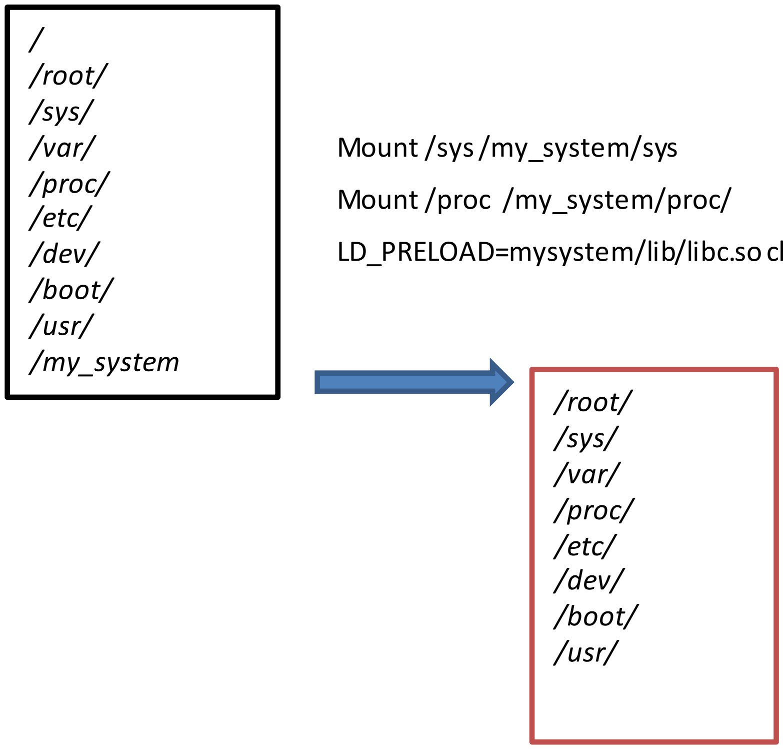
Disable Interrupts in user space	In VxWorks programmer tend to disable interrupts to implement critical sections. Do not that in Linux.
drivers are not in user space	Many embedded devices interact with various types of hardware. VxWorks device drivers are very different from Linux. It is important to re-implement these drivers in Linux.
Virtual address space	Non Linux RTOS access addresses physically. In Linux this is done differently.
Ioperm	Accessing IO ports in Linux must be authorized when doing so in user space.
Kill task	Programmers usually move all tasks to a single processes that shares all addresses. But killing a single thread is not possible in Linux.
Semaphores behavior	Linux Posix semaphores are inferior to VxWorks they do not support Priority.
Zero copy	Accessing physical ram to and from any device is not done directly in Linux.
Access to executables symbols	VxWoks offers the user an easy way to modify variables inside a running task. This is not so easy on Linux. Use dl functions family.
Windows build environment	Though seem unrelated, the human factor is biggest issue when it comes to porting. Programmers would like to keep their eclipse. It is possible to develop and debug in windows to Linux using eclipse.
Latency	Do not expect to get the same latency. VxWorks beats Linux. I measured a context switch in ATOM based Linux is average of 9us. In vxWorks it was 1us.
Number of priorities	While Linux has 100 vxWorks has 256.
Abundance of features	Use open source. Do no re-invent. The state of mind of closed-source programmer is to code.

	In Linux you modify and configure.
Programming languages	Not all has to be c/c++. There are programming languages in Linux that may easily execute under real time constraints.
Licensing	Make sure you know what sort of open source license you assimilate into the code.
Multicore	Simply use it Linux. Isolate processors, route IRQS and so on.
Other architectures	Linux runs in various processors architectures, ARM , SH Power and so on. Make sure you test these hardware with cyclictest before choosing the hardware.
Do not assume all is runnable on all architectures	There are many bundles out there. Qt, openCV, OpenGL and so on. Make sure they run on the target machine. Use the yocto the company suggests.

8.4 From Linux to Linux – Fast & Dirty

It may be required to from time to time to port the product from your embedded Linux to some other Linux based operating system. This may not be as easy as one would first imagine and may take a while.

Many times, it is just needed to check how well the product performs in the client's hardware and his operating system. Therefore; it would be best to simply run your software in a contained environment without any changes at all. So, if possible we suggest to use the "chroot" command. By copying the entire root file system of the product there is no need to compile or do any modifications; i.e. the product is the same binary as long as the processor and the kernel can execute it.



```
Mount /sys /my_system/sys
Mount /proc  /my_system/proc/
LD_PRELOAD=mysystem/lib/libc.so chroot bash
```

9. Interoperability

Interoperability is the capability of different systems to work together without being specially configured to do so.

In addition to their main objectives, embedded devices have many other functionalities such as recovery, logging user interface, configuration etcetera. User interfaces come in various ways, many times a manufacturer will provide a web interface or an executable binary GUI. Either way, interoperability here means that these interfaces may be in Laptops, tablets, smart phones smart glasses, IoT sensors and so on. Though the interface is not the main core of the product, it is an essential part of it and may be considered it as its “front face”.

Therefore, companies spend lots of resources in this area and due to that embedded programmers spend portions of their time adding interoperability to the product. Engineers needs to define the interfaces (WEB? GUI ?), the mediums (Ethernet, WIFI, Bluetooth …), what parts of the software to expose (security) and many other aspects which are beyond the scope of this book.

However, since interoperability functionalities are so common there are many protocols that it is wise to re-use them in various forms. For instance, for easy discovery and configuration upnp is very useful.

If you have a raspberry PI and a usb camera please refer to the Surveillance camera exercise.

10. Logging

In Linux it is customary to launch executables in the background. The term used for putting a process in the background is “to daemonize” or to launch as a “service”. When a process is “daemonized” we cannot see its standard output so it is common to use logging. Logging is unseparated part of the code.

The common API for logs is syslog. Most logging functions have priorities and some have tags. This enables us to control which logs are being printed in real time and which do not.

Add to netcat.c syslog facility.Print all inputs.
Test it.

11. Debugging

Linux is abundant with debugging and monitoring tools. In the following sections, we will show some of these tools.

Process	**top latencytop ps htop gdb kernelshark valgrind oprofile strace duma gcc stap njamd , pmap**
memory	**free vmstat slabtop**
Processor	**mpstat cpustat perf dstat vmstat /proc/interrupts**
Net	**Ifconfig bmon tcpdump wireshark ip nicstat ping**
Disk	**iostat blktrace dtrace iotop**

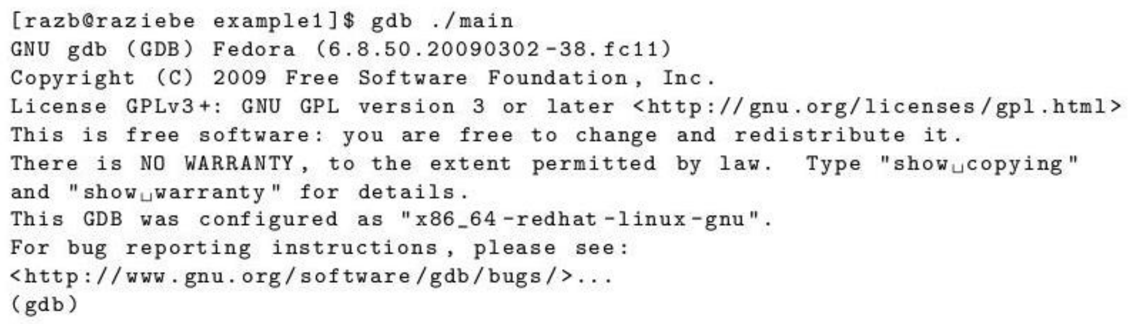
11.1 GDB

The Gnu debugger is a source level debugger. Gdb has various GUI skins like eclipse and ddd. This section presents the textual user interface (TUI).

To be able to debug a program that program has to be compiled with the –g flag (or in other variations). To debug it enter the command:

```
$ gdb <executable>
```

For example: to debug a program named main.

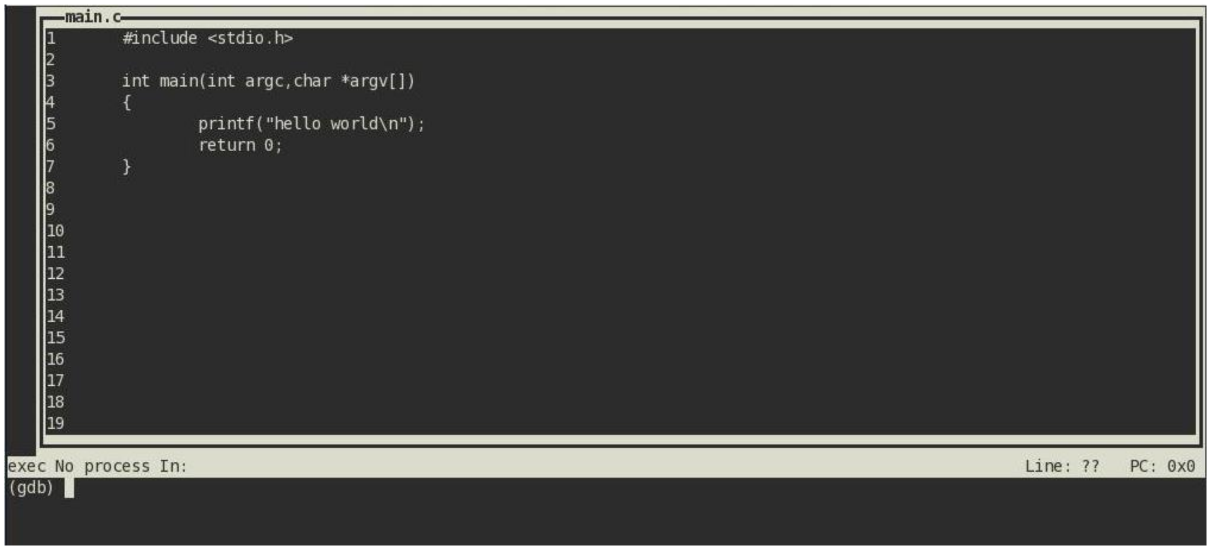


```
[razb@raziebe example1]$ gdb ./main
GNU gdb (GDB) Fedora (6.8.50.20090302-38.fc11)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show␣copying"
and "show␣warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
(gdb)
```

If you enter the command “list” you will get the source code.

11.1.1 Interface

Gdb has textual interface (TUI) . To use it please enter: *ctrl a+x+a* or simply use –*tui* argument when running gdb; i.e. gdb -tui.



```
main.c
        #include <stdio.h>

        int main(int argc,char *argv[])
        {
                printf("hello world\n");
                return 0;
        }
exec No process In:                                              Line: ??    PC: 0x0
(gdb)
```

11.1.2 Basic Commands

To start debugging enter the start commands.

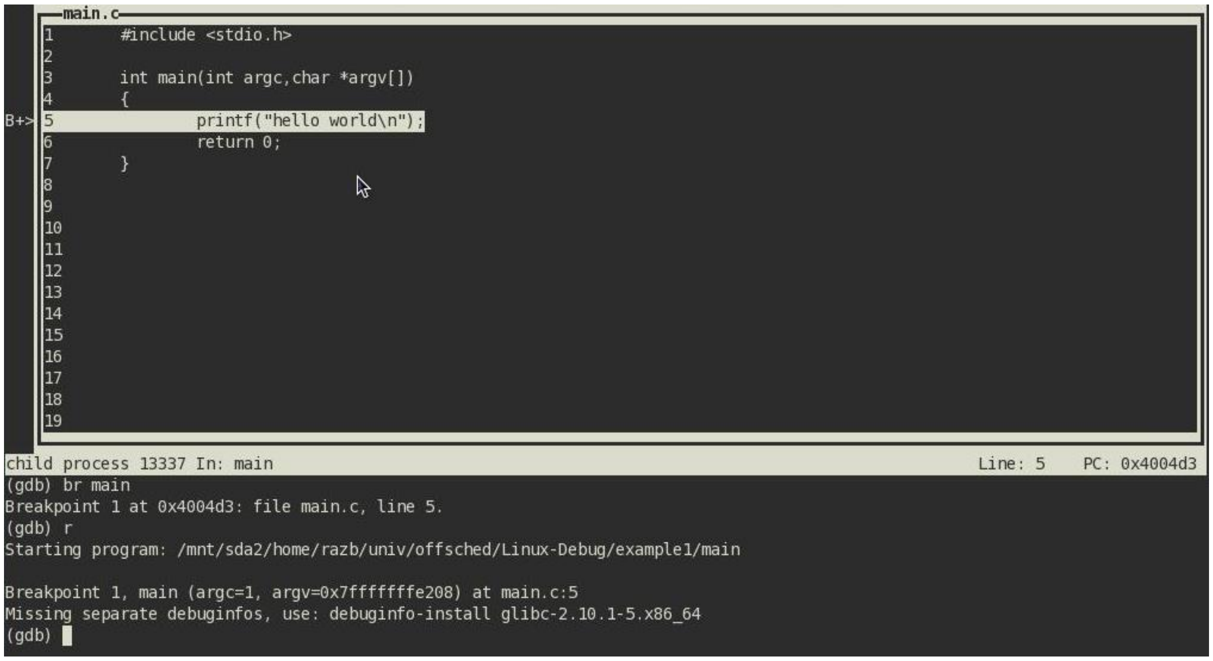
```
(gdb) start
```

If the program has arguments put them after the run or start command.

```
(gdb) start arg1 arg2
```

To insert a break point in main enter:

```
$ br main
```



```
main.c
        #include <stdio.h>

        int main(int argc,char *argv[])
        {
B+>             printf("hello world\n");
                return 0;
        }
```

```
child process 13337 In: main                                    Line: 5    PC: 0x4004d3
(gdb) br main
Breakpoint 1 at 0x4004d3: file main.c, line 5.
(gdb) r
Starting program: /mnt/sda2/home/razb/univ/offsched/Linux-Debug/example1/main

Breakpoint 1, main (argc=1, argv=0x7fffffffe208) at main.c:5
Missing separate debuginfos, use: debuginfo-install glibc-2.10.1-5.x86_64
(gdb) 
```

The program stopped at the first command in main(). The sign B marks that this is a break point. To move use the "next "command (A shortcut is "n") or "step" (A shortcut is "s"). When entering "n" over a function we walk over the function, when entering "s" over a function we enter the function. To examine the values, enter the command "print <value>" command. To view a variable's type, use the command "ptype <variable>".

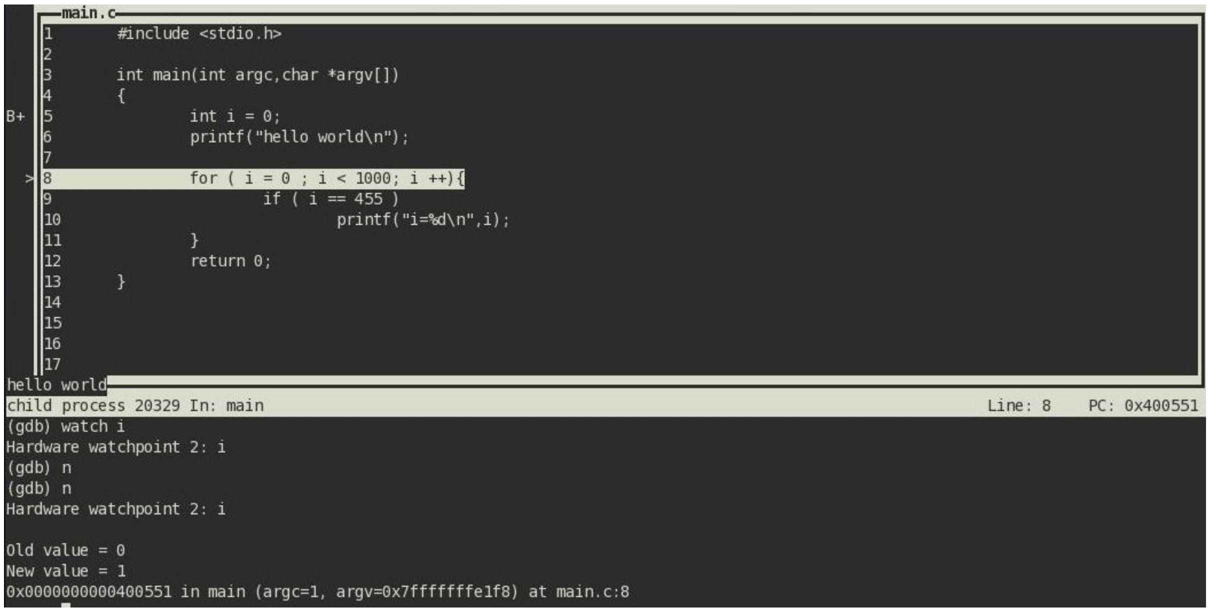
To examine the program's threads, use the command "info threads". To move from one thread to another thread use the command "thread

<id>”. To apply a certain command on a thread use the command “thread <command>”.

11.1.3 Watch Points

A watch point is an observation command on a variable. For instance, when we want to know if and when a variable gets a certain value. Watch points can be supported by the hardware or not. The difference is in the speed of the execution of the process. To check whether the hardware supports hardware watch points enter:

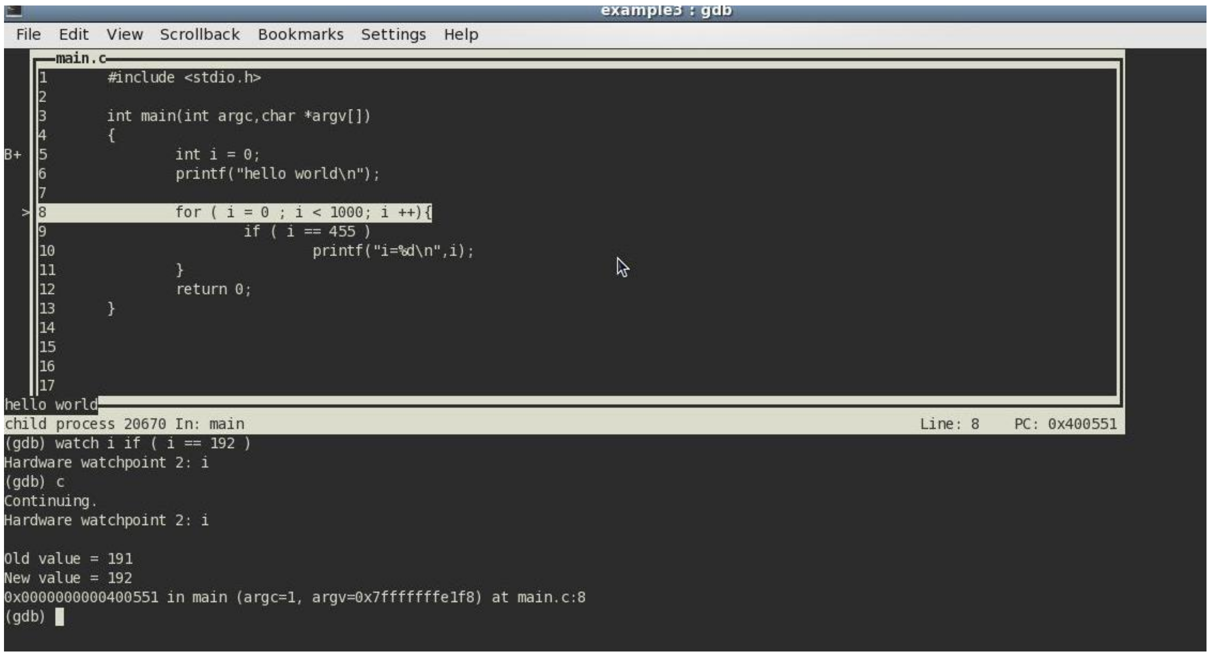
(gdb) *show can-use-hw-watchpoin*t.



```
main.c
        #include <stdio.h>

        int main(int argc,char *argv[])
        {
B+              int i = 0;
                printf("hello world\n");

 >              for ( i = 0 ; i < 1000; i ++){
                        if ( i == 455 )
                                printf("i=%d\n",i);
                }
                return 0;
        }




hello world
child process 20329 In: main                                            Line: 8     PC: 0x400551
(gdb) watch i
Hardware watchpoint 2: i
(gdb) n
(gdb) n
Hardware watchpoint 2: i

Old value = 0
New value = 1
0x0000000000400551 in main (argc=1, argv=0x7fffffffe1f8) at main.c:8
```

Suppose we want to check if “i” ever equals 192. Enter the command:

```
$ watch i if ( i==192)
```

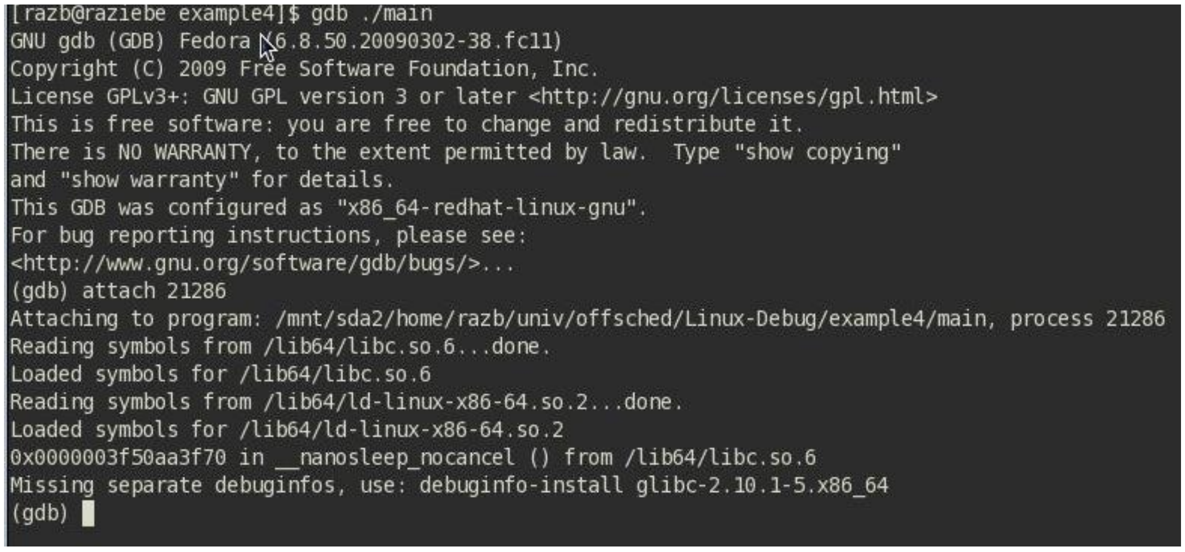


```
        #include <stdio.h>

        int main(int argc,char *argv[])
        {
B+              int i = 0;
                printf("hello world\n");

  >             for ( i = 0 ; i < 1000; i ++){
                        if ( i == 455 )
                                printf("i=%d\n",i);
                }
                return 0;
        }
```

```
hello world
child process 20670 In: main                                        Line: 8    PC: 0x400551
(gdb) watch i if ( i == 192 )
Hardware watchpoint 2: i
(gdb) c
Continuing.
Hardware watchpoint 2: i

Old value = 191
New value = 192
0x0000000000400551 in main (argc=1, argv=0x7fffffffe1f8) at main.c:8
(gdb)
```

11.1.4 Attach a running process

To attach to a running process enter the attach command and after the process id (pid). In the picture bellow, we attach process pid 21286.



```
[razb@raziebe example4]$ gdb ./main
GNU gdb (GDB) Fedora (6.8.50.20090302-38.fc11)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
(gdb) attach 21286
Attaching to program: /mnt/sda2/home/razb/univ/offsched/Linux-Debug/example4/main, process 21286
Reading symbols from /lib64/libc.so.6...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x0000003f50aa3f70 in __nanosleep_nocancel () from /lib64/libc.so.6
Missing separate debuginfos, use: debuginfo-install glibc-2.10.1-5.x86_64
(gdb)
```

11.1.5 Post mortem debugging

Post mortem debugging in gdb is possible by using a core dump. A Core dump is file generated when a process crashes. It is a snapshot of the process state when the process was killed by the kernel. The crash is usually a segmentation violation error. The kernel has to be instructed to generate a core dump. To instruct the kernel to create a core dump use the "ulimit" command as follows:

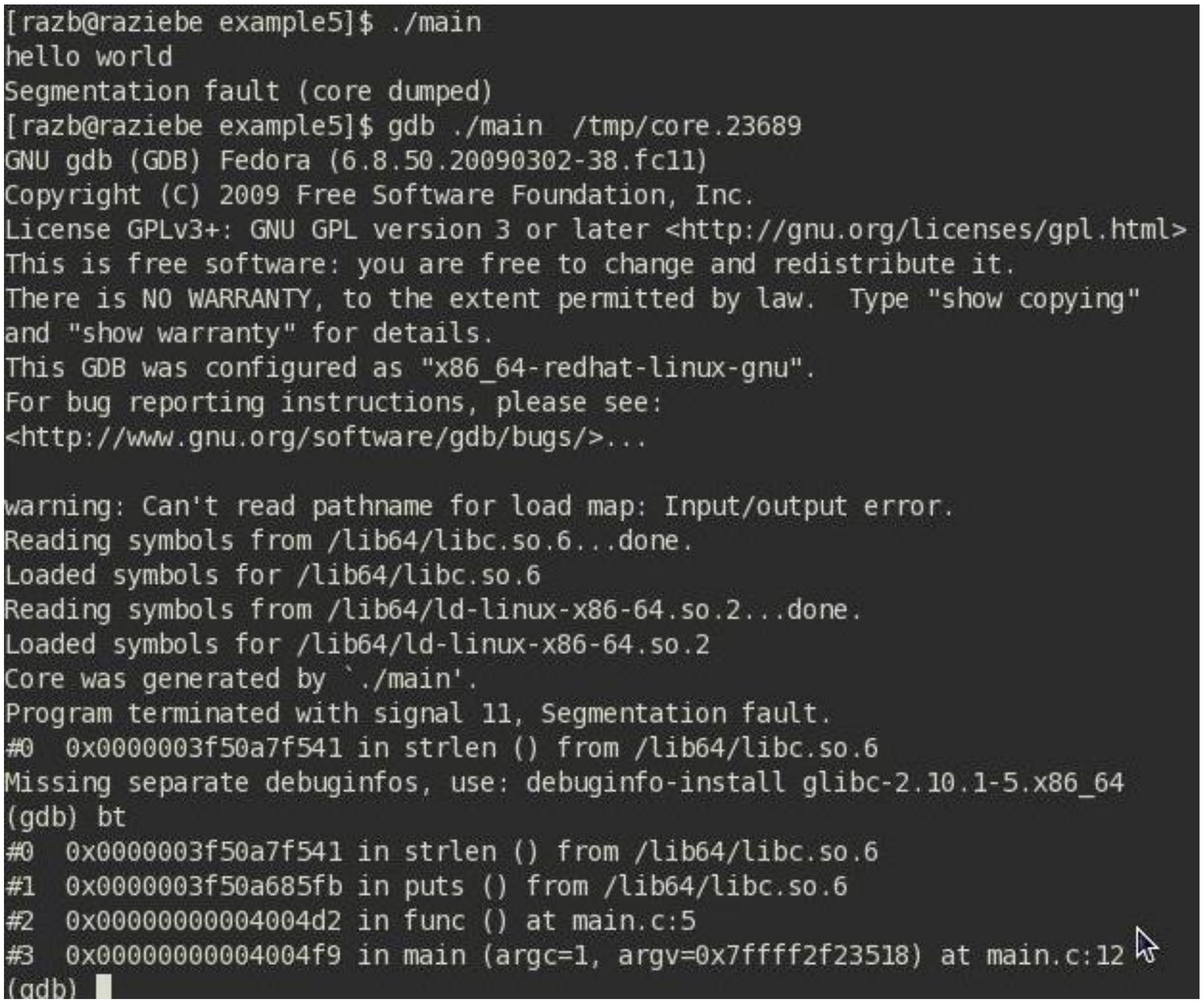
```
$ ulimit –c unlimited
```

For example:

```
#include <stdio.h>

Int func(void)
{
        Printf("%s\n",NULL);
}

int main(int argc, char* argv[])
{
        int i;
        printf(" hello world\n");
        func();
}
```

Compile and run this program. It will crash and say that a core has been dumped. Usually the core file is written to the current directory. It is possible to change the core file directory by changing the **core pattern**.

```
$ echo /tmp/core.%p > /proc/sys/kernel/core_pattern
```

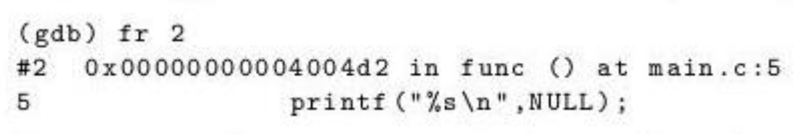


```
[razb@raziebe example5]$ ./main
hello world
Segmentation fault (core dumped)
[razb@raziebe example5]$ gdb ./main  /tmp/core.23689
GNU gdb (GDB) Fedora (6.8.50.20090302-38.fc11)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib64/libc.so.6...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Core was generated by `./main'.
Program terminated with signal 11, Segmentation fault.
#0  0x0000003f50a7f541 in strlen () from /lib64/libc.so.6
Missing separate debuginfos, use: debuginfo-install glibc-2.10.1-5.x86_64
(gdb) bt
#0  0x0000003f50a7f541 in strlen () from /lib64/libc.so.6
#1  0x0000003f50a685fb in puts () from /lib64/libc.so.6
#2  0x00000000004004d2 in func () at main.c:5
#3  0x00000000004004f9 in main (argc=1, argv=0x7ffff2f23518) at main.c:12
(gdb) 
```

In this case the core file was directed to the /tmp/ directory. To examine the core file, use the command: gdb <executable> <core file>.

For example:

```
$ gdb ./main /tp/core.23689
```



```
(gdb) fr 2
#2  0x00000000004004d2 in func () at main.c:5
5                printf("%s\n",NULL);
```

And here it is – the failure.

11.1.6 Loading shared libraries

When a debugged a program has parts of its binary code in shared objects we need to tell gdb for the whereabouts of these shared objects (also known as shared libraries or dlls). To instruct gdb to present the shared objects search path use the command:

```
<gdb> show solib-search-path
To change the search path, enter:
<gdb> set solib-search-path <path1>:<path2>
For example, to add /tmp/shared_objects/ to the search path:
<gdb> set solib-search-path /tmp/shared_object
```

11.1.7 Gdb user command

It is possible to define a debug command. The syntax is as follows:

```
<gdb > define myusrcmd
        print $arg0
        end
```

Then we can apply this command in some break point, for example, consider the code bellow:

```
    #include <stdlib.h>
    #include <mcheck.h>
    #include <malloc.h>

    int main(void)
    {
            int i =0,j=0 ;
            for (;i<100;i++){
                  j++;
           }
    }
```

We can add the command:

```
(gdb ) define test1
             if $arg0 == 99
             call puts("arg is 99")
             end
```

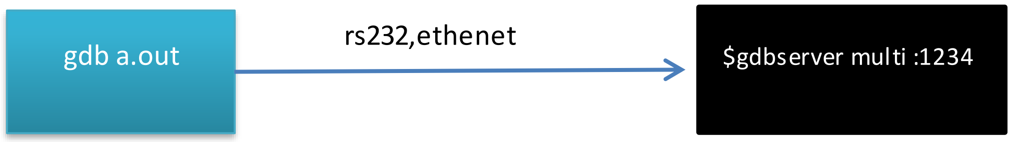
And then assign it to a break point to it as follows:

```
(gdb) br 11

Breakpoint 1 at 0x4004c8:  file MDBG.c, line 11.
(gdb)  command 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>test1
> continue
> end
```

Run gdb.

11.1.8 Remote GDB

A cross platform development is when we build on one computer and execute the program in another computer. To debug a program executed in another computer we execute a gdbserver on the target computer and gdb on the build machine.



The steps to debug a binary called a.out are as follows:

Create a stripped (no debug symbols) copy of the binary and keep the original binary a.out with debug symbols. Then copy the stripped binary to target computer. We strip the binary only to reduce the size of binary we copy. In the bellow example, the target computer ip is 172.16.2.100.

```
$ strip a.out -o a.out.strip
$ scp a.out.strip 172.16.2.100:/tmp/a.out
```

Now login to the target computer and execute the following commands:

```
$ gdbserver –multi :1234 /tmp/a.out
$ gdb a.out
…
Reading symbols from /home/razpc/opensource/linuxdebug-code/glibc/mallinfo/a.out...done.
```

Now we connect to the target gdbserver from the local gdb, for this we use the “target remote” command.

```
(gdb) target remote 172.16.2.100:1234
Remote debugging using 172.16.2.100:1234
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x00007f857ee125b0  in ?? () from /lib64/ld-linux-x86-64.so.2
```

In Android emdedded, it is possible to debug by assigning an IP to the android target machine (usb tethering).

11.1.9 GDB Non Stop Mode

In case you are executing a multithreaded application and wish to debug a single thread without stopping the rest of the threads use the following configuration commands.

Enable the async interface.

```
(gdb)    set target-async 1
```

If we use CLI gdb, pagination breaks non-stop.

```
(gdb)   set pagination off
```

Finally, turn it on !

```
(gdb)  set non-stop on
```

11.2 Memory Failures

Memory failures are considered failures that are hard to detect. This section details the various methods and tools used in Linux.

Libc defines three types of memory allocations:

1. **Static allocation.** Part of the C language. Refers to static and global variables.

2. **Automatic allocation.** Part of the C language. Local variables.

3. **Dynamic allocation.** Not part of the C language. It refers to malloc implementation and is part of the GNU C library.

11.3 Valgrind

Valgrind is a sort of cpu simulation. It runs a program line by line and checks the correctness of the line. It is important to note that valgrind does not find all errors and it slows the program upto factor of 50. It does not execute multithreaded application in parallel but serializes threads. Nonetheless, Valgrind is a powerful debugging tool.

Valgrind suite includes various tools, memchek, profile and so on. The default is memcheck.

The following sections details how various error types are detected by Valgrind.

11.3.1 Memory Corruption

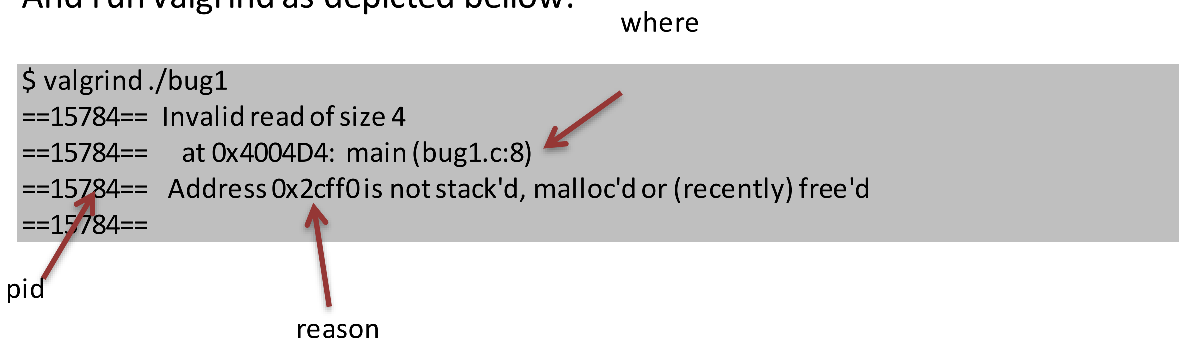
Here is a short program that demonstrates how Valgrind responds to segmentation error.

```
int main()
{
        int x;
        int *p=0x2cff0;

        x = *p;
}
```

Compile it. Make sure to use –g.

```
$ gcc -g bug1.c -o bug1
```

And run valgrind as depicted bellow.



```
$ valgrind ./bug1
==15784==  Invalid read of size 4
==15784==    at 0x4004D4:  main (bug1.c:8)
==15784==   Address 0x2cff0 is not stack'd, malloc'd or (recently) free'd
==15784==
```

11.3.2 Memory Leaks

Valgrind memory checker also helps to find memory leaks. When the examined program exits a report is emitted. This report details the state of each memory block. A memory block is said to be:

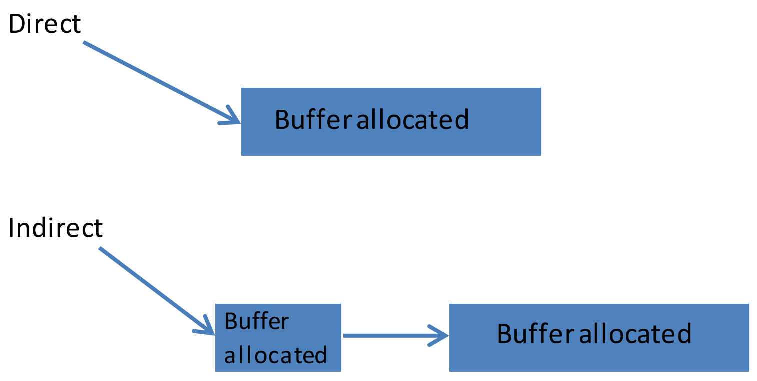
Definitely lost: These blocks leak

Indirectly lost: These blocks leak

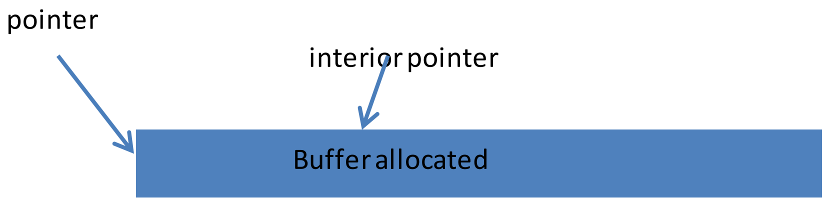
Possibly lost: It is possible that these blocks leak

Still reachable: These blocks do not leak.

To understand what all this mean, we need to understand what a direct and indirect mean, and what an interior pointer means.



An indirect lost pointer means that the parent pointer is lost. A lost pointer means a parent pointer is released.



Interior pointer points at the middle of a block.

Definitely lost

Please refer to the directory “direct” in the course for an example.

```
int main()
{
            someClassA* x = new someClassA;
}
```

Still reachable

Variable “p” is still reachable because the pointer is not changed

```
char *p;

int main()
{
       p  = (char *)malloc(11);
}
```

Possibly lost example

It is possible that the buffer is lost. This is because “t” may or may-not be what the programmer intended.

```
char *t;

int main()
{
	t = (char *)malloc(100);
	t+=4;
}
```

Indirect lost example

Indirect lost means that the blocks that point to the block are lost.

```
class A{
	char *__p;
	char *__q;

public:
	void foo();
	A(void);
	~A(){ free(__p);	}
};


A::A()
{
	__p = (char *)malloc(sizeof(char));
	__q = (char *)malloc(sizeof(char));
}

int main(void)
{
		A* x = new A;
}
```

.

```
razpc@razpc:~/opensource/valgrind/indirect$ valgrind --leak-check=full ./bug
==4060== Memcheck, a memory error detector
==4060== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==4060== Using Valgrind-3.6.1-Debian and LibVEX; rerun with -h for copyright info
==4060== Command: ./bug
==4060==
blabla
==4060==
==4060== HEAP SUMMARY:
==4060==     in use at exit: 18 bytes in 3 blocks
==4060==   total heap usage: 3 allocs, 0 frees, 18 bytes allocated
==4060==
==4060== 1 bytes in 1 blocks are definitely lost in loss record 2 of 3
==4060==    at 0x4C28F9F: malloc (vg_replace_malloc.c:236)
==4060==    by 0x400621: A::A() (bug.cpp:29)
==4060==    by 0x40066B: main (bug.cpp:36)
==4060==
==4060== 17 (16 direct, 1 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 3
==4060==    at 0x4C28B35: operator new(unsigned long) (vg_replace_malloc.c:261)
==4060==    by 0x400660: main (bug.cpp:36)
==4060==
==4060== LEAK SUMMARY:
==4060==    definitely lost: 17 bytes in 2 blocks
==4060==    indirectly lost: 1 bytes in 1 blocks
==4060==      possibly lost: 0 bytes in 0 blocks
==4060==    still reachable: 0 bytes in 0 blocks
==4060==         suppressed: 0 bytes in 0 blocks
==4060==
==4060== For counts of detected and suppressed errors, rerun with: -v
==4060== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 4 from 4)
```

What happened here? Please explain. Will we have indirect lost pointer if we delete the parent?

11.3.3 Overlap

An overlap means that service routines like strncpy or memcpy are having the source override the destination.

```
void foo(char *src, char *dest, int len)
{
        strncpy(dest, src, len);
}

int main(void)
{
        char *s1 = (char *)malloc(16);
        char *s2 = s1 +5;

        strcpy(s1,"hello ");
        foo(s1, s2, 6);
        return 0;
}
```

Valgrind reports overlap error.

```
==4345==  Source and destination overlap in strncpy(0x51d004b, 0x51d0046,  6)
==4345==    at 0x4C299B6: strncpy (mc_replace_strmem.c:339)
==4345==    by 0x40056F: foo(char*, char*, int) (bug.cpp:11)
==4345==    by 0x4005BF: main (bug.cpp:20)
```

11.3.4 Illegal frees

An illegal free is when free gets an illegal pointer.

```
int foo(int *x)
{
	free(x);
}

int main(void)
{
	int *x = malloc(sizeof(int));
	*x = 4;
	x++;
	printf("hello world\n");
	foo(x);
}
```

And the output.

```
==4377== Invalid free() / delete / delete[]
==4377==    at 0x4C282E0: free (vg_replace_malloc.c:366)
==4377==    by 0x40059B: foo (bug.c:10)
==4377==    by 0x4005D8: main (bug.c:19)
==4377==  Address 0x51d0044 is 0 bytes after a block of size 4 alloc'd
==4377==    at 0x4C28F9F: malloc (vg_replace_malloc.c:236)
==4377==    by 0x4005AF: main (bug.c:15)
==4377==
```

11.3.5 Valgrind V-bits

V-bits stands for virtual bits. Every variable in the program has a virtual presentation. Each time a program variable is changed then its virtual presentation is changed as well. Yet, is important to note that copying variables around does not cause memcheck to check/report errors. Only if the variable is used in such a way that affects the program behavior Valgrind checks for errors.

Consider the following code (in ext2 directory):

Will Valgrind report errors?

```
int i,j;
int a[10],b[10];

for (i = 0 ; i < 10; i++ ){
      j = a[i];
      b[i] = j;
}
if ( j == 19)
      return -1;
```

And now? why ?

```
int i,j;
int a[10],b[10];

for (i = 0 ; i < 10; i++ ){
      j = a[i];
      b[i] = j;
}
```

11.3.6 Definedness

In the previous exercise the command “j += a[i]” is not erroneous from Valgrind perspective. The “undefinedness” of j is not **observable**. Only when j is checked Valgrind complains.

Consider the bellow code (ex3). What Valgrind reports? Why?

```
struct my_struct {
   int x;
   char z;
};

struct my_struct T1;
struct my_struct T2;

   int *x1, *x2;
   struct the_struct T1;
   struct the_struct T2;

   T1.x = 1;
   T1.z = '0';

   T2 = T1;
   x1 = (int *) &T2.z;
   x2 = (int *) &T1.z;
   if (*x1 == *x2)
      printf("u.y=%d,%d\n", T1.z, T2.z);
```

11.3.7 Writing suppression files

There are times where we wish to suppress some of Valgrind errors; for example, when these errors come from third party software. The suppress rules are gathered in a suppression file and Valgrind helps us create suppression files.

Example:

```
    int *x;
     x = new (int);
     x++;
     *x = 0;
```

When running this code with Valgrind we get:

```
==2280== Invalid write of size 4
==2280==    at 0x400583: main (bug.cpp:13)
==2280==  Address 0x5971044 is 0 bytes after a block of size 4 alloc'd
==2280==    at 0x4C28B35: operator new(unsigned long) (vg_replace_malloc.c:261)
==2280==    by 0x400575: main (bug.cpp:11)
==2280==
```

Now let say we want to ignore it for some reason, for this we execute valgrind with –gen-suppressions flag.

```
$ valgrind --gen-suppressions=yes ./a.out
==2280== Invalid write of size 4
==2280==    at 0x400583: main (bug.cpp:13)
==2280==  Address 0x5971044 is 0 bytes after a block of size 4 alloc'd
==2280==    at 0x4C28B35: operator new(unsigned long) (vg_replace_malloc.c:261)
==2280==    by 0x400575: main (bug.cpp:11)
==2280==
==2280==
==2280== ---- Print suppression ? --- [Return/N/n/Y/y/C/c] ----y
{
   <insert_a_suppression_name_here>
   Memcheck:Addr4
   fun:main
}
```

The part inside the parenthesis is the suppress rule. Just copy it into a suppression rules file and use the –suppressions=<filename> flag.

11.3.8 Common Warnings

We will not cover all messages, but only the “weird” ones.

Client switching stacks

The program had moved the stack pointer in more than 2000000 bytes. Valgrind believes that the program is switching to another thread so it tries to move the base stack pointer to a different location.

Noted but unhandled ioctl <number>

An undocumented/wrapped ioctl has been made.

11.4 Duma

Duma is an open source GPL malloc() memory debugging service. It is a fork from electric fence (efence is considered obsolete).

Duma detects:

1. Over-runs and under-runs for C and C++.
2. Access freed memory.
3. Help detect some memory leaks

Duma/efence methodology

When an incorrect malloc()'ed memory access (an underrun or an overrun) is made the process may not get a segmentation violation by the processor. Duma is meant to force the program to get a violation error.

First download duma from https://sourceforge.net/projects/duma/files/latest/download and build it.

Duma comes with some examples, I will demonstrate on example1.cpp.

There are several ways to use duma. First one is to link duma directly to your application.

```
$ g++ example1.cpp -c -g -o example1.o
$ g++ example1.o -pthread libduma.a -o ./example
$ ./example
DUMA 2.5.15 (static library)
Copyright (C) 2006 Michael Eddington <meddington@gmail.com>
Copyright (C) 2002-2008 Hayati Ayguen <h_ayguen@web.de>, Procitec GmbH
Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

Segmentation fault (core dumped)
```

And now to analyze the failure we simply debug the core dump.

```
raz@ubuntu01:~/linuxdebug-code/malloc_debugger/duma_2_5_15$ gdb example core
GNU gdb (Ubuntu 7.10-1ubuntu2) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from example...done.

warning: core file may not match specified executable file.
[New LWP 8838]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Core was generated by `./e'.
Program terminated with signal SIGSEGV, Segmentation fault.
```

```
#0  0x0000000000400e43 in main () at example1.cpp:12
12          pi[i] = i;
```

Another way is to use the LD_PRELOAD bash shell environment without linking duma to your application.

```
$ g++ example1.cpp –g –o example1
$ LD_PRELOAD=./libduma.so.o ./example
```

Duma (and electric fence) overrides malloc (it does not use the malloc hook) and for this reason we need to LD_PRELOAD it. If we want to run the program with gdb we do not use LD_PRELOAD because gdb fails to override malloc. For this reason, to use duma with Gdb we need to link duma to our application.

Example1 present a common overrun bug. If you run example1 binary you should see a notice of a core dump of gdb report of access violation.

11.5 Glibc

Glibc provides some facilities for the programmer to debug malloc'ed memory corruption and memory leaks.

11.5.1 Memory trace with mtrace

The bellow code presents mtrace. Mtrace is a memory tracer. It traces all memory allocations and de-allocations from when mtrace is called until when muntrace is called.
For example:

```
#include <malloc.h>

int main(void)
{

        mtrace(); /* Starts the recording of memory allocations and releases */
        int* a;
        a = malloc(sizeof(int)); /* allocate memory and assign it to the pointer */
        muntrace();
}
```

Now compile this program:

```
$ gcc –g mtrace.c
```

Run it as follows:

```
$ MALLOC_TRACE=malloc.trace ./a.out
```

A text file called malloc.trace is created. You can parse it, but an easy tool called mtrace can help parse it. Here is an example output:

```
$ mtrace a.out malloc.trace
0x0000000000b78460      0x4  at mtrace.c:9
```

11.5.2 Memory corruption checker

Glibc mprobe (void *pointer) checks if there's memory corruptions of the given pointer. The bellow example show how glibc detects memory under-run.

```
#include <stdlib.h>
#include <mcheck.h>
#include <malloc.h>

int main(void)
{
    mcheck_status s;
    int *a = NULL;

    a = (int *)malloc(4);
    a[-1] = 1;
    s = mprobe(a);
    switch (s) {
    case MCHECK_DISABLED:
        puts("MCHECL_DISABLE");
        break;
    case MCHECK_OK:
        puts("MCHECK_OK:");
        break;
    case MCHECK_HEAD:
        puts("MCHECK_HEAD:");
        break;
    case MCHECK_TAIL:
        puts("MCHECK_TAIL");
        break;
    case MCHECK_FREE:
        puts("MCHECK_FREE");
        break;
   }
}
```

Compile it and link it to mcheck.

```
$ g++ –g –lmcheck
$ ./a.out
```

```
memory clobbered before allocated block
Aborted
```

Mprobe returns values are:

MPROBE_DISABLED – did not compile with –mprobe

MPROBE_OK – no corruption detected.

MPROBE_HEAD – underrun

MPROBE_TAIL – overrun

MPROBE_FREE – block is free

There are cases where mprobe does not return but reports an error.

11.5.3 Another Memory corruption checker

Another way to debug malloc'd memory corruptions is through the environment variable **MALLOC_CHECK_.**

This technique uses different kind of mechanism which is independent to mcheck. Its advantage is in the fact that there is no need to link against mcheck. Yet it is considered inferior to mcheck.

In the example bellow we overrun a buffer.

```
    a = (int *)malloc(4);
    a[1]  =1;
    free(a);
```

Compiling this code and then running the following line:

```
$ MALLOC_CHECK=1  ./a.out
*** glibc detected *** ./a.out: free(): invalid pointer: 0x0000000002092010   ***
```

11.5.4 Mallinfo

It is possible to get a Malloc information through two APIs, mallainfo or malloc_stats. mallinfo is useful whenever we wish to query for memory leaks. See the example bellow (in glibc directory).

```
void dumpmallinfo(void)
{
        struct mallinfo info;

        info = mallinfo();
        printf("\n\n%s\n"
                "Bytes allocated with sbrk by malloc %d\n"
                "Chunks not in use %d\n"
                "Bytes allocated with mmap %d\n"
                "Memory occupied by malloc chunks %d\n"
                "Memory occupied by free chunks %d\n"
                ,prefix,
                info.arena,
                info.ordblks,
                info.hblkhd,
                info.uordblks,
                info.fordblks);
}
...
        dumpmallocinfo("Pre Any Allocation");
        a = malloc(sizeof(int)); /* allocate memory and assign it to the pointer */
        if (a == NULL) {
                return 1; /* error */
        }

        dumpmallocinfo("after single malloc of 4 bytes");
        free(a);
        dumpmallocinfo("after release of previous malloc");

        for (i = 0 ; i< 10;i++){
                p[i] = malloc(101);
        }
        dumpmallocinfo("after allocation of 10 101-bytes buffers");
        for (i = 0 ; i< 9;i++){
                free(p[i]);
        }
        dumpmallocinfo("after release of 9 101-bytes buffers");
```

Here is the output of the above program.

```
Pre Any Allocation
Bytes allocated with sbrk by malloc 0
Chunks not in use 1
Bytes allocated with mmap 0
Memory occupied by malloc chunks 0
Memory occupied by free chunks 0


after single malloc of 4 bytes
Bytes allocated with sbrk by malloc 135168
Chunks not in use 1
Bytes allocated with mmap 0
Memory occupied by malloc chunks 32
Memory occupied by free chunks 135136


after release of previous malloc
Bytes allocated with sbrk by malloc 135168
Chunks not in use 1
Bytes allocated with mmap 0
Memory occupied by malloc chunks 0
Memory occupied by free chunks 135168

after allocation of 10 101-bytes buffers
Bytes allocated with sbrk by malloc 135168
Chunks not in use 1
Bytes allocated with mmap 0
Memory occupied by malloc chunks 1120
Memory occupied by free chunks 134048


after release of 9 101-bytes buffers
Bytes allocated with sbrk by malloc 135168
Chunks not in use 1
Bytes allocated with mmap 0
Memory occupied by malloc chunks 112
Memory occupied by free chunks 135056
```

11.5.5 The Malloc hook

The malloc hook is a useful tool to monitor all memory allocations made by the process. The programmer is expected to replace the original malloc with his own allocator, he may choose to allocate from a designated pool or add some useful reports and back traces.

Read malloc_hook man page. Replace malloc with mmap over anonymous memory or memalign.

11.6 Gcc memory checks

After the coding is done, the compiler is almost the last barrier before the bug. So, it would be smart to use this tool as best we can. First, it is a good practice to compile programs with –Wall or –Werror, or even with –pedantic or –pedantic-errors.

The followings are other tricks that can help reduce memory bugs.

11.6.1 Gcc stack Protector

GCC provides a useful feature that help debugging stack corruptions. The option is called –fstack-protector and it is enabled by default in most distributions, but it is not enough because it does not guard against all functions.

Consider for example the bellow corruption:

```
int array[10];
int i;

for (i=0;i<=10;i++)
    array[i]=0;
```

This piece of code is not detected by valgrind and obviously not by mtrace. Yet luckily GCC provides a useful feature that checks the stack for corruption. Use the option –fstack-protector-all for that and execute the code:

```
$ gcc  array.c -g -fstack-protector-all  && ./a.out
*** stack smashing detected ***: ./a.out terminated
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(__fortify_fail+0x37)[0x7f063c57e4f7]
/lib/x86_64-linux-gnu/libc.so.6(__fortify_fail+0x0)[0x7f063c57e4c0]
./a.out[0x4005b2]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xed)[0x7f063c4a530d]
./a.out[0x400469]
======= Memory map: ========
00400000-00401000  r-xp 00000000  08:01  4326092                    /
```

The address of the crash

11.6.2 Automatic Allocation

In cases when we allocate a block that is used only in function block, that is in braces open mark "{" and the braces close mark "}", you can use variable length for allocating blocks. For example:

```
int foo(int len)
{
	int array[len]; /* automatic allocation */
}
```

Prefer automatic allocation over malloc.

11.7 Memory Overcommit

Thanks for the virtual memory a process can allocate more RAM memory than the machine has. The size of the overcommitted memory is in **/proc/sys/vm/overcommit_memory**. To disable "memory overcommit":

```
$ echo 2 > /proc/sys/vm/overcommit_ratio.
```

11.8 Oom Killer

The **Out of memory killer** is a mechanism embedded into the Linux kernel. It is activated whenever the system is in a memory stress. The kernel decides which process to kill according to an oom score of each process. The higher the value in **/proc/<pid>/oomadj** the bigger the chances it would be killed. Setting 17 into **/proc/<pid>/oomadj** prevents the process from being killed. The oom_score in /proc/<pid> determines how bad a process is. Oom_score is a function of oomadj and the duration of the process.

11.9 Live Debugging

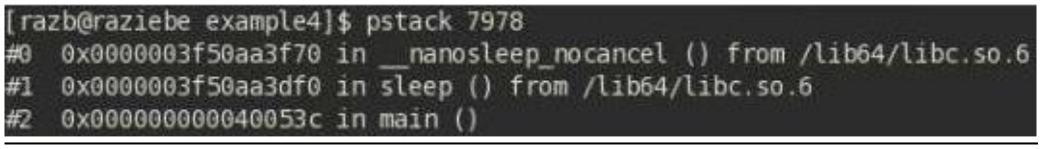
There are many times that neither can we stop nor wish to stop an executing program. However; we may want to gather information about the process. There are ways to collect information about the process without stopping or re-running it.

11.9.1 Pstack

Pstack command is used to dump a running process stack.

```
# include <stdio.h>

int main ( int argc , char * argv [])
{
	int i = 0;
	printf ( " hello world \ n " );
	for ( i = 0 ; i < 1000; i ++){
		sleep (1);
	}
	return 0;
}
```

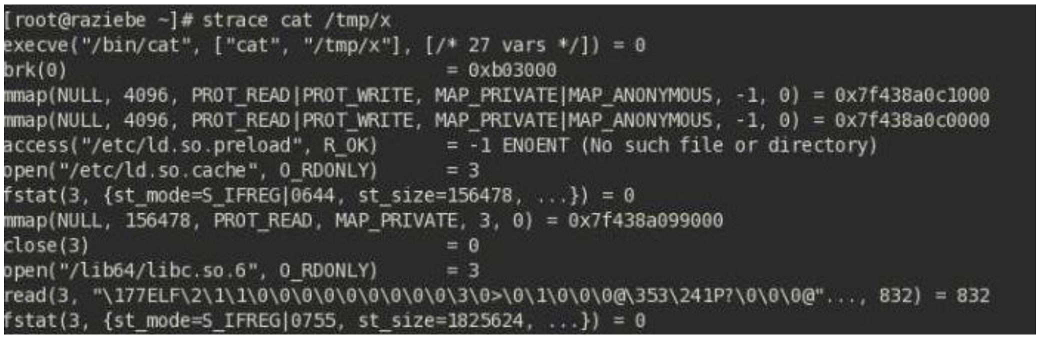
To print the stack of process with 7978, which executes the above source code, enter pstack <pid>.



```
[razb@raziebe example4]$ pstack 7978
#0  0x0000003f50aa3f70 in __nanosleep_nocancel () from /lib64/libc.so.6
#1  0x0000003f50aa3df0 in sleep () from /lib64/libc.so.6
#2  0x000000000040053c in main ()
```

11.9.2 Strace

Strace traces all system calls of running a process. For example, the bellow command dumps all system calls of the process “cat”.



```
[root@raziebe ~]# strace cat /tmp/x
execve("/bin/cat", ["cat", "/tmp/x"], [/* 27 vars */]) = 0
brk(0)                                  = 0xb03000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f438a0c1000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f438a0c0000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=156478, ...}) = 0
mmap(NULL, 156478, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f438a099000
close(3)                                = 0
open("/lib64/libc.so.6", O_RDONLY)      = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@\353\241P?\0\0\0@"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1825624, ...}) = 0
```

Strace can attach to a running process with the –p <pid> argument.

11.9.3 Backtrace

backtrace() returns the stack of the calling thread. Here is an example:

```
static void dump_trace ( void )
{
        int j, ptrs;
        int len = 32;
        void *buffer[len];
        char **s;

        ptrs = backtrace(buffer, len);
        s = backtrace_symbols(buffer, ptrs);
        if (s == NULL)
                return;
        for (j = 0; j < ptrs; j++)
                printf("%s\n", s[j]);
}
```

Make sure you compile the code with rdynamic.

In this exercise we know we have in one of our c structs a memory leak.
For example:

```
struct A{
            char x;
};

struct B{
            short x;
};

Struct C{
            int x;
};
```

Add a malloc hook that dumps allocations stack of struct C only.

11.10 A Process back door - libcli

Libcli is administration software that provides a telnet-like command line interface into a program. Libcli provides a telnet server thread that executes inside your program.

Though it appears not as a debug tool, it is actually very useful. We will demonstrate some uses for it later. But first let us download it. Currently it is hosted on git hub.

```
$ git clone https://github.com/dparrish/libcli.git
```

This command downloads the sources as a repository. Let's build it:

```
$ cd libcli

$ make
```

Build output is libcli.so and a test program named clitest.

Let use clitest and demonstrate how to use it.

```
$ LD_LIBRARY_PATH=./ ./clitest
```

Open a new session and enter:

```
$ telnet localhost 8000
```

You will see few lines as bellow:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
libcli test environment
Username:
```

The user name is fred and password is nerk . Once you entered the login you'll see:

```
router>
```

if you enter “help”, you’ll get a list of commands.

```
Commands available:
  help              Show available commands
  quit              Disconnect
  logout            Disconnect
  exit              Exit from current mode
  history           Show a list of previously run commands
  enable            Turn on privileged commands
  test
  show regular      Show the how many times cli_regular has run
  show counters     Show the counters that the system uses
  show junk
  debug regular     Enable cli_regular() callback debugging
  context           Test a user-specified context
```

Add a malloc hook to your program and implement an overrun memory boundary checker. Do that by using mprotect. Then add libcli a control command which activates and deactivates it. Start with libcli/test1.c

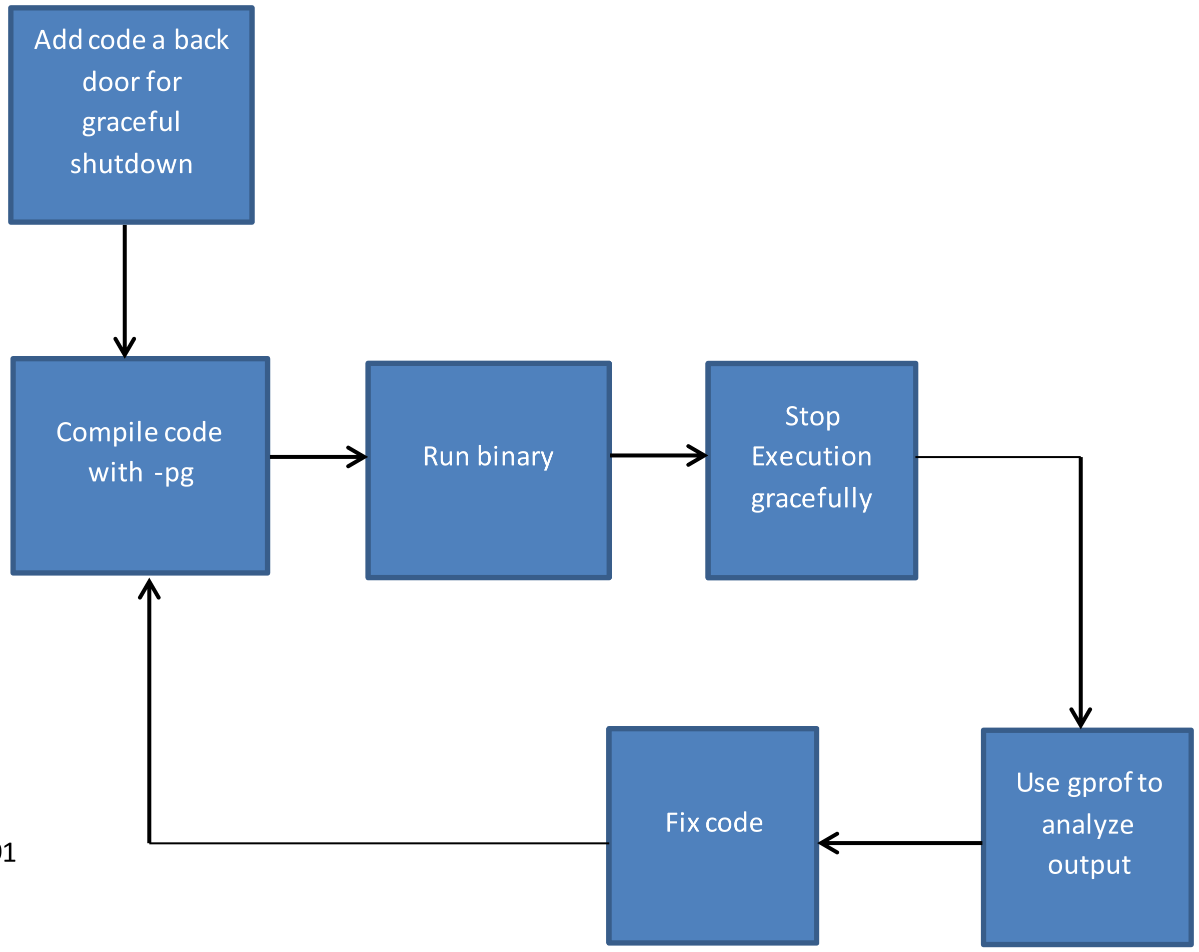
Add a facility to print a certain value upon a cli command.

12. Profiling

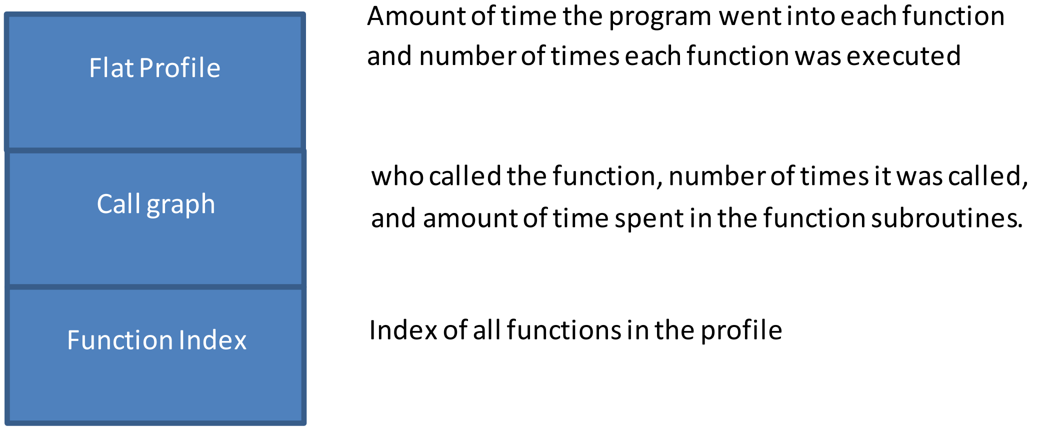
The common use of profiling is to measure the amount of time a group of commands consume.

12.1 Gprof

Gprof is the GNU profiler embedded in the GCC compiler. To use it use the "–pg" compiler option. The –pg option is used to generate an output file called gmon.out to be used by gprof – gnu profiler.


Add code a back door for graceful shutdown
Compile code with -pg
Run binary
Stop Execution gracefully
Use gprof to analyze output
Fix code

When running Gprof, it prints the information in three forms:


Flat Profile
Amount of time the program went into each function and number of times each function was executed
Call graph
who called the function, number of times it was called, and amount of time spent in the function subroutines.
Function Index
Index of all functions in the profile

Gprof main command line options are:

-A print annotated (The code with profiling information) source code

-b brief. Gprof will print not explanation of the profile

-c print with the call graph

-I (capital i) <dirs> source directory search path

-l line profiling

-p flat profile

Let us take a short example. The bellow is a code that saves all square roots up to some range in an array (see the Sqrt directory).

```
#include <math.h>
#include <pthread.h>
#include <stdio.h>

#define RANGE 1000000

float results[RANGE];
pthread_mutex_t mtx =  PTHREAD_MUTEX_INITIALIZER;

float calc_sqrt(float n){  return sqrt(n);}
void my_lock(void){        pthread_mutex_lock(&mtx);}
void my_unlock(void){   pthread_mutex_unlock(&mtx);}

int printSqrt(int range)
{
            int j =0;
            float i = 0;
            float sq;
            int sq1;

            for (; i < range; i++) {
                        sq = calc_sqrt(i);
                        sq1 = sq;
                        if (sq == sq1) {
                                    my_lock();
                                    results[(int)i] = sq;
                                    my_unlock();
                                    continue;
                        }
                        my_lock();
                        results[(int)i] = -1;
                        my_unlock();
            }
}
int main(){         printSqrt(RANGE);}
```

Compile this program with –pg –g. once done, enter:

```
$ gprof –b a.out
```

Output should be similar the bellow.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %    cumulative  self             self    total
 time   seconds   seconds   calls   ms/call ms/call name
100.92    0.02     0.02   1000000    0.00    0.00   calc_sqrt
  0.00    0.02     0.00   1000000    0.00    0.00   my_lock
  0.00    0.02     0.00   1000000    0.00    0.00   my_unlock
  0.00    0.02     0.00         1    0.00   20.18   printSqrt

                         Call graph

granularity: each sample hit covers 2 byte(s) for 49.55% of 0.02 seconds

index  % time   self  children      called          name
                0.02    0.00    1000000/1000000     printSqrt [2]
[1]     100.0   0.02    0.00        1000000        calc_sqrt [1]
-----------------------------------------------
                0.00    0.02      1/1                main [3]
[2]     100.0   0.00    0.02      1                printSqrt [2]
                0.02    0.00 1000000/1000000          calc_sqrt [1]
                0.00    0.00 1000000/1000000          my_lock [4]
                0.00    0.00 1000000/1000000          my_unlock [5]
-----------------------------------------------
                                                 <spontaneous>
[3]    100.0    0.00    0.02                        main [3]
                0.00    0.02      1/1                printSqrt [2]
-----------------------------------------------
                0.00    0.00   1000000/1000000      printSqrt [2]
[4]     0.0     0.00    0.00   1000000             my_lock [4]
-----------------------------------------------
                0.00    0.00   1000000/1000000      printSqrt [2]
[5]     0.0     0.00    0.00   1000000            my_unlock [5]
-----------------------------------------------

Index by function name
```

Average time spent in printSqrt and descendants

Calc_sqrt run for 0.02 secs. The rest ran for less them 0.01sec

%time: % from tot time
Self: tot time in function
Children: time in children

printSqrt is a parent of calc_sqrt. Calc_sqrt contributed 0.02 secs to the printSqrt that propagated to main()

Don't know who is the parent

Flat Profile

Call graph

```
[1] calc_sqrt          [5] my_unlock
[4] my_lock             [2] printSqrt
```

Where is the contention?

Profile the Matrix example (in gprof directory). Find out where the contentions are and reduce them.

12.1.1 Gprof Instrumentation Internals

The –pg flag instructs the compiler to add a call to a function called mcount usually at the beginning of a frame.

The bellow should output "call mcount" to terminal

```
$ echo 'main(){}' | gcc -x c -S  -pg -o - - | grep mcount
```

Explanation:

The first "|" creates a file with just an empty "main" and prints it to the standard output.

The second "|" instructs the compiler to collect input from stdin and dump the output to stdout. The "–x c " tells the compiler that this is a c code, -S tells it not to assemble, -pg means to add profiling data.

You can add override mcount by writing mcount yourself. The bellow is what one expects in a tracing function:

```
int __attribute__ ((no_instrument_function)) mcount(int ip,int parent_ip)

{       if (trace_flags && TRACE_ME)

                        my_trace ();

}
```

The attribute no_instrument_function is necessary else mcount calls itself. Ip and parent_ip are the address of the calling frame and the line after that.

12.2 GCC instrumentation

A GCC option “–finstrument-functions” adds two new functions which are called each time any regular function is called. The functions signature is:

```
__cyg_profile_func_enter(void *this_fn, void *call_site);
__cyg_profile_func_exit(void *this_fn, void *call_site);
```

The prefix cyg stands for Cygnous. Cygnous is the company that added this feature to gcc.

These functions can be used in any way. Here is an example:

```
void __attribute__((__no_instrument_function__))
   __cyg_profile_func_enter(void *this_fn, void *call_site)
{
           printf("Enter function 0x%X called from 0x%X\n",
                      (unsigned long)this_fn, (unsigned long) call_site);
}

void __attribute__((__no_instrument_function__))
   __cyg_profile_func_exit(void *this_fn, void *call_site)
{
           printf("Exit function 0x%X called from 0x%X\n",
                      (unsigned long)this_fn, (unsigned long)call_site);
}
```

Extend cli code.
Add a command to libcli that dumps the stack of each thread on demand by using –pg.

12.3 Oprofile

Oprofile is a profiler used both for user space programs and kernel space code. The way this profiler works is by making a snapshot of the executing context. The snapshot is taken in a non-mask-able interrupt (NMI) each time this interrupt is executing. The interval for which the NMI is woken (the sampling rate) needs not to be neither too short not too long.

Oprofile is composed from some user space tools, a module and an optional GUI interface.

To operate oprofile please install it:

```
$ sudo apt-get install oprofile
```

And optionally you can install a gui interface.

```
$ sudo apt-get install oprof_start
```

Now, make sure oprofile is compiled into the kernel. If you are using a main distributor kernel it is likely that oprofile is compiled into it.

Login as root.

This command erases previous profiling data.

```
$ opcontrol –reset
```

To start profiling we must initialize:

```
$ opcontrol –deinit

$ oprcontrol –init
```

Now, we need to prepare the profiling configuration. Profiling configuration includes:

The type of we event want to check. For example:

```
$ opcontrol –event CPU_CLK_UNHALTED
```

To watch all available events use:

```
$ opcontrol  –list-events
```

The sampling rate, for example:

```
$ opcontrol  –event:DATA_MEM_REFS:30000
```

Will set the counter reset value to 30000 with event DATA_MEM_REFS.

The "session-dir" is where the output directory of the oprofile. For example:

```
$ opcotrol  –session-dir=/home/raz/myprofile/
```

By default, oprofile data it collected to a single file, we can choose to separate data to different files by cpu or kernel or lib or thread or all.

For example:

```
$ opcontrol –separate=cpu
```

For example the bellow enables 10 frames stack.

```
$ opcontrol –callgraph=10
```

Image argument is used to profile a single task. For example, profile only myapp. If you add separate=lib then the library will appear also in the profiling.

```
$ opcontrol  –image=myapp
```

To profile the kernel you need to use –vmlinux as path to the vmlinux elf, not the vmlinuz.

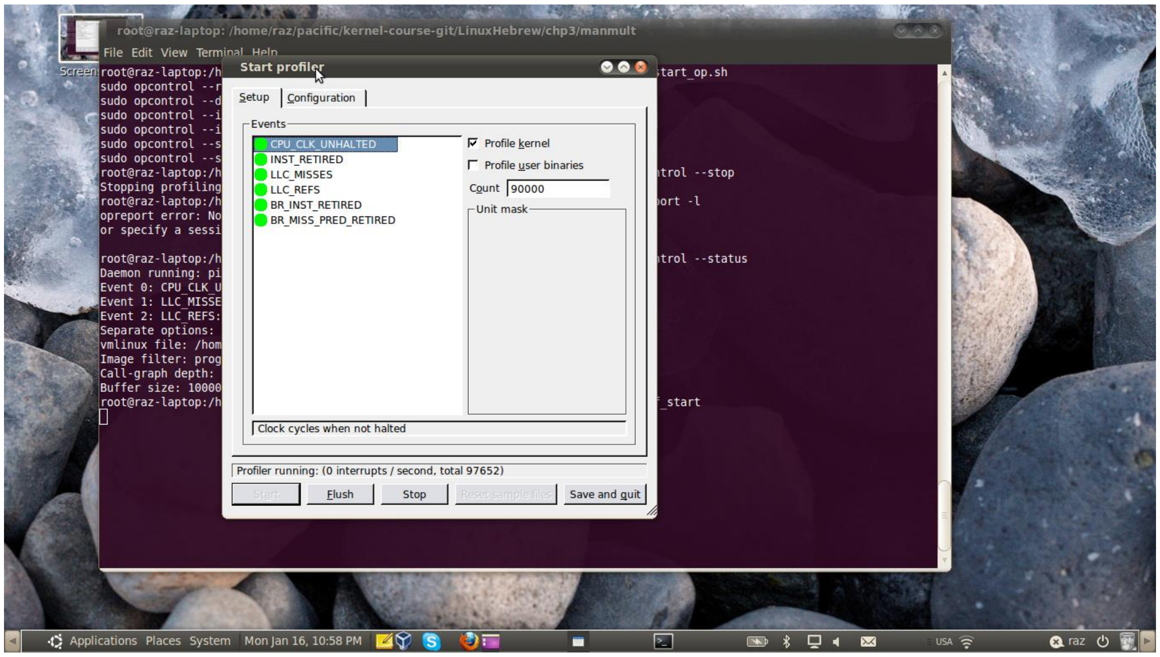
```
$ opcontrol –vmlinux=/boot/vmlinux-2.6.38
```

This is the reason we need to compile a full kernel when profiling modules.

Once you have created the configuration, you can examine what the configuration is now with:

```
$ opcontrol  --status
```

Most of the configuration can be done with oprof_start.



```
root@raz-laptop: /home/raz/pacific/kernel-course-git/LinuxHebrew/chp3/manmult

root@raz-laptop:/h                                        start_op.sh
sudo opcontrol --r
sudo opcontrol --d
sudo opcontrol --i
sudo opcontrol --i
sudo opcontrol --s
sudo opcontrol --s
root@raz-laptop:/h                                        ntrol --stop
Stopping profiling
root@raz-laptop:/h                                        port -l
opreport error: No
or specify a sessi

root@raz-laptop:/h                                        ntrol --status
Daemon running: pi
Event 0: CPU_CLK_U
Event 1: LLC_MISSE
Event 2: LLC_REFS:
Separate options:
vmlinux file: /hom
Image filter: prog
Call-graph depth:
Buffer size: 10000
root@raz-laptop:/h                                        f_start
```

Start profiler

Setup | Configuration

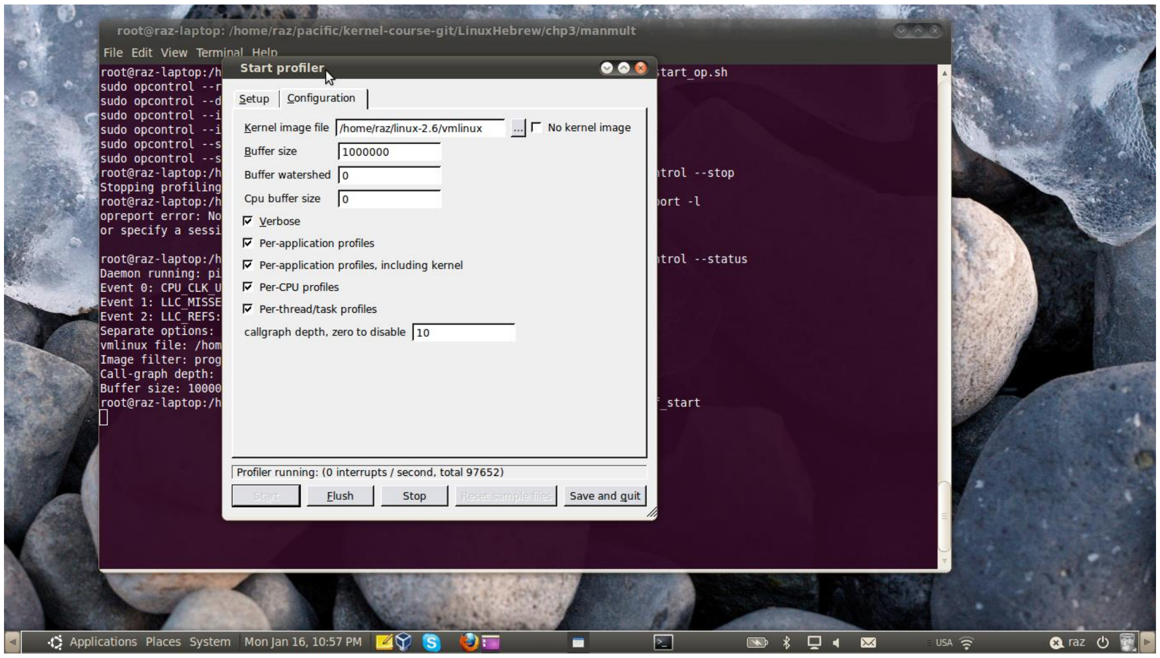
Events
- CPU_CLK_UNHALTED
- INST_RETIRED
- LLC_MISSES
- LLC_REFS
- BR_INST_RETIRED
- BR_MISS_PRED_RETIRED

[✓] Profile kernel
[ ] Profile user binaries
Count 90000
Unit mask

Clock cycles when not halted

Profiler running: (0 interrupts / second, total 97652)

File Edit View Terminal Help
Start
Flush
Stop
Reset sample files
Save and quit
Applications Places System Mon Jan 16, 10:58 PM


Start profiler

Kernel image file: /home/raz/linux-2.6/vmlinux
☐ No kernel image
Buffer size: 1000000
Buffer watershed: 0
Cpu buffer size: 0
☑ Verbose
☑ Per-application profiles
☑ Per-application profiles, including kernel
☑ Per-CPU profiles
☑ Per-thread/task profiles
callgraph depth, zero to disable: 10

Profiler running: (0 interrupts / second, total 97652)

oprof_start view

Either way, configuration is saved in ~/.oprofile/daemonrc.

12.3.1 Test case 1

In oprofile/testcase1 you will find sources file. Build them.

```
$ make
```

Output is called matrix and it is a binary that multiply matrices.

We wish to profile matrix processor consumption. So we use CPU_CLK_UNHALTED event, we also wish to check memory access so we use LLC_MISSES.

```
$ opcontrol –event=CPU_CLK_UNHALTED  --event:LLC_MISSES –
image=test –session-dir=/home/raz/myprofile
```

Then launch oprofile use the script start_op.sh

```
$ opcontrol --start
```

Execute test (do NOT user root user when running test)

```
$ ./test
```

You can also launch oprofile after test was launched. To stop the profiling (whether test still runs or not) enter:

```
$ opcontrol --stop
```

Now let us examine the result. To check the result we use opreport and opannotate.

```
$ opcontrol  --session-dir=/home/raz/oprofile/
```

And we have:

```
CPU_CLK_UNHALT...|LLC_MISSES:466500|
  samples|      %|  samples|      %|
------------------------------------
      943 100.000         1 100.000 prog
        CPU_CLK_UNHALT...|LLC_MISSES:466500|
          samples|      %|  samples|      %|
        ------------------------------------
              709 75.1856         1 100.000 prog
              234 24.8144         0       0 libc-2.11.1.so
```

```
$ opcontrol -l --session-dir=/home/raz/oprofile
```

Output should look like the bellow:

```
CPU: Intel Architectural Perfmon, speed 933 MHz (estimated)
Counted CPU_CLK_UNHALTED  events (Clock cycles when not halted
Counted LLC_MISSES  events (Last level cache demand requests
samples  %          samples  %         image name   symbol name
707      74.9735  0   0          prog              RandomMatrix
234      24.8144  0   0          libc-2.11.1.so    /lib/libc-2.11.1.so
2         0.2121  1   100.000  prog              MatrixCreate
```

What is your second conclusion?

Let’s continue.

This is the output of:

```
$opreport –c –session-dir=/home/raz/profile
```

The –c is call graph.

```
samples  %        samples  %        image name               symbol name
-------------------------------------------------------------------------------
  707      100.000  0               0  prog                       main
707      74.9735  0               0  prog         RandomMatrix
  707      100.000  0               0  prog         RandomMatrix [self]
-------------------------------------------------------------------------------
234      24.8144  0           0  libc-2.11.1.so   libc-2.11.1.so
  234      100.000  0           0  libc-2.11.1.so     libc-2.11.1.so [self]
-------------------------------------------------------------------------------
  2        100.000  1        100.000  prog                 main
2          0.2121  1        100.000  prog                 MatrixCreate
  2        100.000  1        100.000  prog                 MatrixCreate [self]
-------------------------------------------------------------------------------
0               0  0               0  prog                        main
  707      99.7179  0               0  prog                        RandomMatrix
  2          0.2821  1        100.000  prog                        MatrixCreate
  0               0  0               0  prog                        main [self]
-------------------------------------------------------------------------------
```

What do you think now? Where is the bottleneck? Let's continue. Now we take out the heavy guns, opannotate.

```
$ opannotate -s --session-dir=/home/raz/oprofile/
```

Output is quite big, let us look at this code snippest.

```
                               for (j = 0; j < m; ++j)
   707 74.9735 0 0: matrix->arr[i][j] = 0.9589 * (random() % 1000001);
```

What do you think the problem is?

12.3.2 Test case 2

In oprofile/testcase2 you will find test.c. Compile it and run it as follows:

```
$ gcc  test.c -g -o test –lpthread
```

Now execute it with “time.

```
$ time ./test
real    0m9.876s
user   0m17.430s
sys    0m0.000s
```

Real means the amount of time process was running – also known as the wall time. User means the amount of time it spent in user space, and sys means the amount of time it spent in kernel space. In the case above user count is higher than real account because it accumulates two or more processors.

There is a severe bug in this code. Use operf or ocount to detect it and fix it.

12.3.3 Test Case 3

Look in the course sources in oprofile/testcase3. Please compile and run the code as follows:

```
$ time ./test
```

Now change the mask to 0x0 and re-run. You will notice that the execution time is much faster. Use ocount or operf to determine what sort of a problem there is in the code and fix it.

12.4 Perf

Perf is profiler built on top of oprofile infrastructure. To build it you have to build a kernel, or if you are using a distribution kernel simply apt-get it (in the ubunto case linux-tool).

```
$ sudo apt-get install linux-tools-common
```

A good use I find for “perf” is the top-like view. Enter:

```
$ perf top
```

This command provides a top-like view in a function level granularity. There are times where there’s a contention in the system and you do not why, “perf top” is very can provide some insight.

```
  PerfTop:     603 irqs/sec  kernel:77.6%  exact:  0.0% [1000Hz cycles], (all, 4 CPUs)
----------------------------------------------------------------------------------------------------------------


        samples  pcnt function                          DSO
        _______  _____ ____________________________     ___________________________________

         156.00  7.6% __ticket_spin_lock                [kernel.kallsyms]
         121.00  5.9% native_read_tsc                   [kernel.kallsyms]
         119.00  5.8% wcswidth                          /lib/x86_64-linux-gnu/libc-2.13.so
         111.00  5.4% iowrite8                          [kernel.kallsyms]
         105.00  5.1% intel_idle                         [kernel.kallsyms]
          97.00  4.8% delay_tsc                         [kernel.kallsyms]
          84.00  4.1% ioread8                           [kernel.kallsyms]
          52.00  2.5% __schedule                        [kernel.kallsyms]
          44.00  2.2% _raw_spin_unlock_irqrestore       [kernel.kallsyms]
          31.00  1.5% _raw_spin_lock_irqsave             [kernel.kallsyms]
          23.00  1.1% __switch_to                        [kernel.kallsyms]
          21.00  1.0% process_output_block               [kernel.kallsyms]
          21.00  1.0% update_curr                       [kernel.kallsyms]
          21.00  1.0% _atomic_dec_and_lock              [kernel.kallsyms]
```

This view was created when i executed “$find /”. We can see there’s a lot of spinlocks and accessing tsc.

Question: Why it the kernel accessing tsc so much?

12.5 Lightweight Profiling

TSC stands for time stamp counter. This a register that increments each processor cycle. Meaning, if the processor is 2GHz, then each tick is ½ nanosecond.

```
#ifdef __i386__
    #define rdtscll(val) __asm__ __volatile__("rdtsc" : "=A" (val))
#else
#define rdtscll(val) do { \
    unsigned int __a,__d; \
    asm volatile("rdtsc" : "=a" (__a), "=d" (__d)); \
    (val) = ((unsigned long)__a) | (((unsigned long)__d)<<32); \
} while(0)
#endif
```

```
...

__s64 t1,t2'
rdtscll(t1);            // start time sample
foo                     // do whatever you want to measure
rdtscll(t2);            // end time sample
```

12.5.1 Kernel Shark

Kernel shark is a useful visualization tool to analyze context switches.

To install kernel shark first install trace-cmd and the kernel shark:

```
$ sudo apt-get install trace-cmd

$ sudo apt-get install kernelshark
```

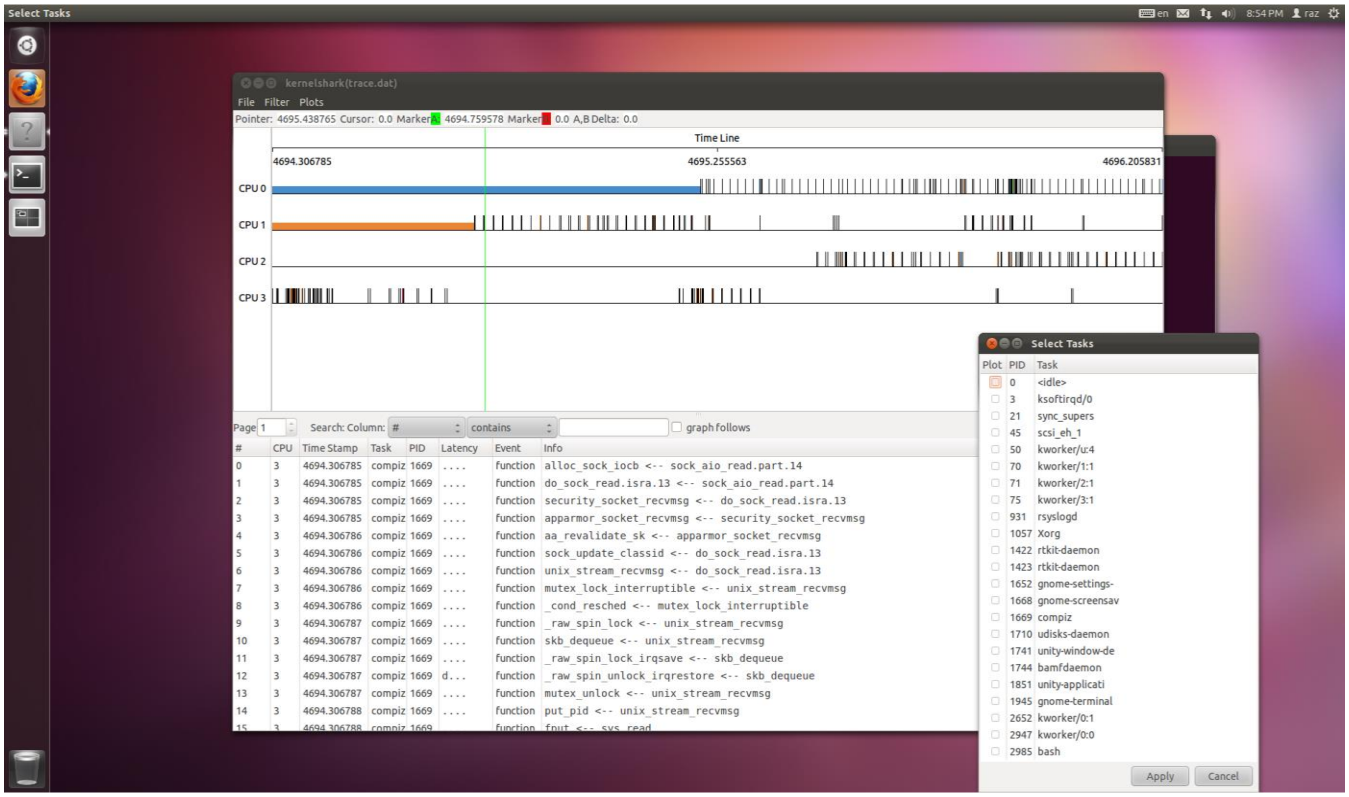
Now, once you create a trace file as shown above , you need to create a dat file, please:

```
$ cd ~/

$ trace-cmd extract
```

The output of this command is trace.dat file. Now launch kernelshark as follows:

```
$ kernelshark trace.dat
```

The bellow is kernelshark.



Pointer: 4695.438765 Cursor: 0.0 MarkerA: 4694.759578 MarkerB: 0.0 A,B Delta: 0.0

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	3	4694.306785	compiz	1669	....	function	alloc_sock_iocb <-- sock_aio_read.part.14
1	3	4694.306785	compiz	1669	....	function	do_sock_read.isra.13 <-- sock_aio_read.part.14
2	3	4694.306785	compiz	1669	....	function	security_socket_recvmsg <-- do_sock_read.isra.13
3	3	4694.306785	compiz	1669	....	function	apparmor_socket_recvmsg <-- security_socket_recvmsg
4	3	4694.306786	compiz	1669	....	function	aa_revalidate_sk <-- apparmor_socket_recvmsg
5	3	4694.306786	compiz	1669	....	function	sock_update_classid <-- do_sock_read.isra.13
6	3	4694.306786	compiz	1669	....	function	unix_stream_recvmsg <-- do_sock_read.isra.13
7	3	4694.306786	compiz	1669	....	function	mutex_lock_interruptible <-- unix_stream_recvmsg
8	3	4694.306786	compiz	1669	....	function	_cond_resched <-- mutex_lock_interruptible
9	3	4694.306787	compiz	1669	....	function	_raw_spin_lock <-- unix_stream_recvmsg
10	3	4694.306787	compiz	1669	....	function	skb_dequeue <-- unix_stream_recvmsg
11	3	4694.306787	compiz	1669	....	function	_raw_spin_lock_irqsave <-- skb_dequeue
12	3	4694.306787	compiz	1669	d...	function	_raw_spin_unlock_irqrestore <-- skb_dequeue
13	3	4694.306787	compiz	1669	....	function	mutex_unlock <-- unix_stream_recvmsg
14	3	4694.306788	compiz	1669	....	function	put_pid <-- unix_stream_recvmsg
15	3	4694.306788	compiz	1669		function	fput <-- sys_read

Select Tasks

Plot	PID	Task
☐	0	<idle>
☐	3	ksoftirqd/0
☐	21	sync_supers
☐	45	scsi_eh_1
☐	50	kworker/u:4
☐	70	kworker/1:1
☐	71	kworker/2:1
☐	75	kworker/3:1
☐	931	rsyslogd
☐	1057	Xorg
☐	1422	rtkit-daemon
☐	1423	rtkit-daemon
☐	1652	gnome-settings-
☐	1668	gnome-screensav
☐	1669	compiz
☐	1710	udisks-daemon
☐	1741	unity-window-de
☐	1744	bamfdaemon
☐	1851	unity-applicati
☐	1945	gnome-terminal
☐	2652	kworker/0:1
☐	2947	kworker/0:0
☐	2985	bash

Why usleep(Tus) is never accurate enough ?

Write a program that executes usleep in a loop, in profiles it with gettimeofday. Wrap usleep with tracing. When the delta is too high stop it and analyze the trace.

Your code should similar to the bellow:

```
while (1) {
                t1<← get time
                usleep(1000);
                t2 ← get time
                if ( t2 – t1 > 1200us )
                        exit();
```

13. System

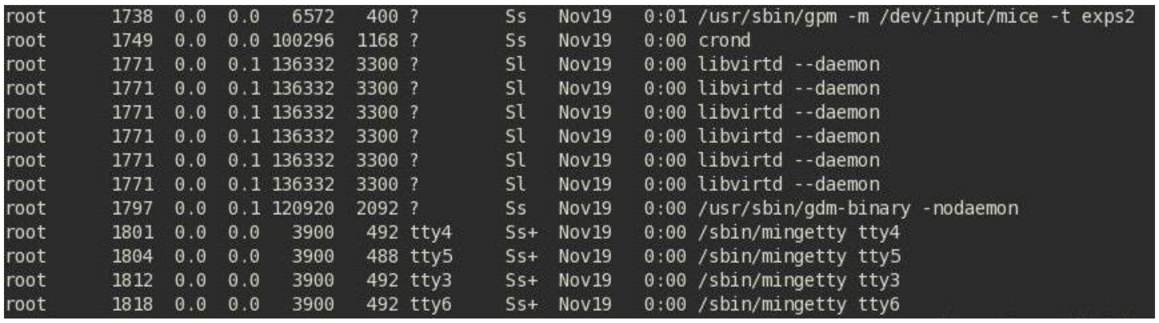
13.1 System Health

13.1.1 ps – List all processes

The “ps” command presents processes in the system. The bellow is an output of the command:

```
$ ps auxH
```

```
1          2     3     4      5       6   7        8     9        10    11
```



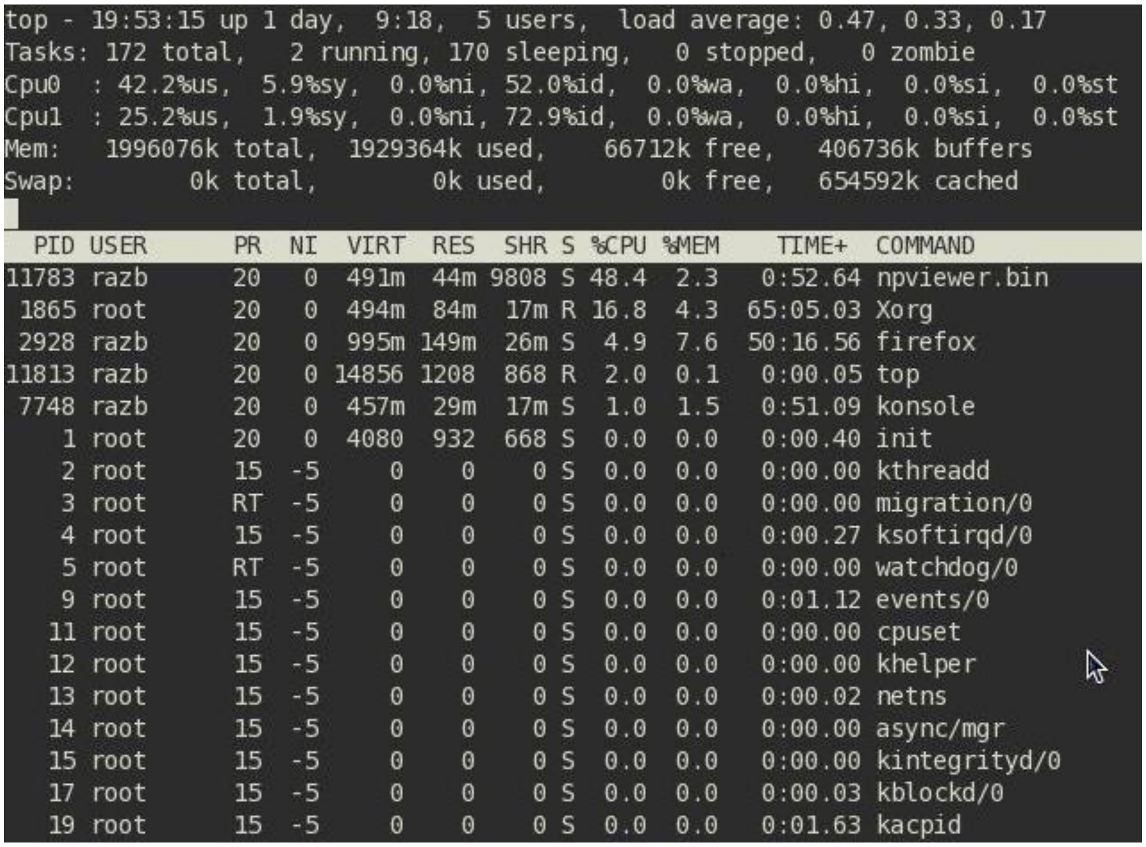
```
root       1738  0.0  0.0   6572   400 ?        Ss   Nov19   0:01 /usr/sbin/gpm -m /dev/input/mice -t exps2
root       1749  0.0  0.0 100296  1168 ?        Ss   Nov19   0:00 crond
root       1771  0.0  0.1 136332  3300 ?        Sl   Nov19   0:00 libvirtd --daemon
root       1771  0.0  0.1 136332  3300 ?        Sl   Nov19   0:00 libvirtd --daemon
root       1771  0.0  0.1 136332  3300 ?        Sl   Nov19   0:00 libvirtd --daemon
root       1771  0.0  0.1 136332  3300 ?        Sl   Nov19   0:00 libvirtd --daemon
root       1771  0.0  0.1 136332  3300 ?        Sl   Nov19   0:00 libvirtd --daemon
root       1771  0.0  0.1 136332  3300 ?        Sl   Nov19   0:00 libvirtd --daemon
root       1797  0.0  0.1 120920  2092 ?        Ss   Nov19   0:00 /usr/sbin/gdm-binary -nodaemon
root       1801  0.0  0.0   3900   492 tty4     Ss+  Nov19   0:00 /sbin/mingetty tty4
root       1804  0.0  0.0   3900   488 tty5     Ss+  Nov19   0:00 /sbin/mingetty tty5
root       1812  0.0  0.0   3900   492 tty3     Ss+  Nov19   0:00 /sbin/mingetty tty3
root       1818  0.0  0.0   3900   492 tty6     Ss+  Nov19   0:00 /sbin/mingetty tty6
```

Column	Description
1	The user who invoked the process
2	The process id
3	CPU usage
4	RAM usage in percentage
5	Virtual Memory usage in percentage
6	RAM consumption (usually anonymous memory)
7	he terminal id of the process

8	Process State: **D** sleeping non-interruptible **R** Process is running (in the run queue) **S** Process is sleeping but interruptible **T** Stopped process **Z** zombie Additional information **<** Not nice – with high priority **N** Nice. Low priority **S** Group leader. (the getsid() return value) **l** Process has threads **+** process in the foreground
9	Invocation date
10	Invocation Time
11	Command used to run the process

13.1.2 Top

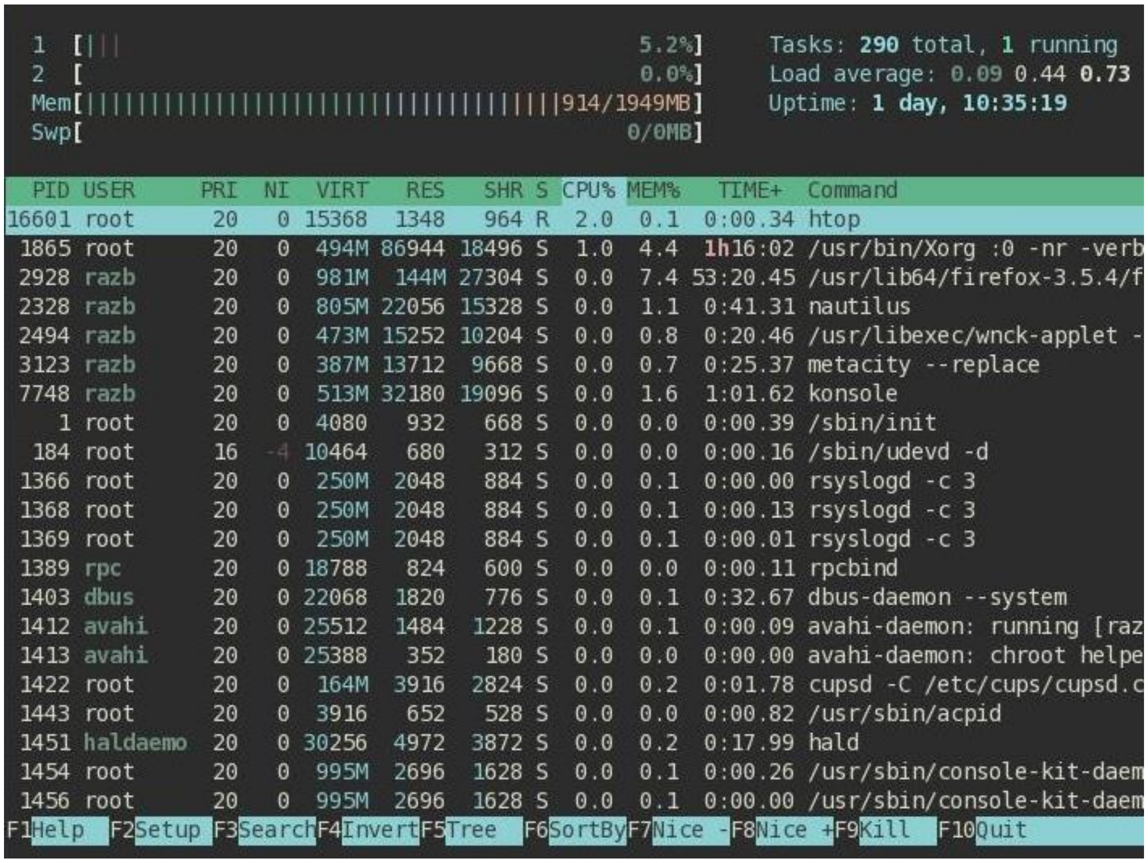
Top is a program that presents all running processes.



```
top - 19:53:15 up 1 day,  9:18,  5 users,  load average: 0.47, 0.33, 0.17
Tasks: 172 total,   2 running, 170 sleeping,   0 stopped,   0 zombie
Cpu0  : 42.2%us,  5.9%sy,  0.0%ni, 52.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  : 25.2%us,  1.9%sy,  0.0%ni, 72.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1996076k total,  1929364k used,    66712k free,   406736k buffers
Swap:        0k total,        0k used,        0k free,   654592k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11783 razb      20   0  491m  44m 9808 S 48.4  2.3   0:52.64 npviewer.bin
 1865 root      20   0  494m  84m  17m R 16.8  4.3  65:05.03 Xorg
 2928 razb      20   0  995m 149m  26m S  4.9  7.6  50:16.56 firefox
11813 razb      20   0 14856 1208  868 R  2.0  0.1   0:00.05 top
 7748 razb      20   0  457m  29m  17m S  1.0  1.5   0:51.09 konsole
    1 root      20   0  4080  932  668 S  0.0  0.0   0:00.40 init
    2 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      15  -5     0    0    0 S  0.0  0.0   0:00.27 ksoftirqd/0
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    9 root      15  -5     0    0    0 S  0.0  0.0   0:01.12 events/0
   11 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 cpuset
   12 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 khelper
   13 root      15  -5     0    0    0 S  0.0  0.0   0:00.02 netns
   14 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 async/mgr
   15 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kintegrityd/0
   17 root      15  -5     0    0    0 S  0.0  0.0   0:00.03 kblockd/0
   19 root      15  -5     0    0    0 S  0.0  0.0   0:01.63 kacpid
```

13.1.3 Htop

Htop is used to display processes like top with friendly interface. Htop provides sorting in various ways. The bellow is list processes sorted by CPU consumption.



```
  1  [|||                                                5.2%]     Tasks: 290 total, 1 running
  2  [                                                   0.0%]     Load average: 0.09 0.44 0.73
  Mem[|||||||||||||||||||||||||||||||||||||||||914/1949MB]     Uptime: 1 day, 10:35:19
  Swp[                                                  0/0MB]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
16601 root       20   0 15368  1348   964 R  2.0  0.1  0:00.34 htop
 1865 root       20   0  494M 86944 18496 S  1.0  4.4  1h16:02 /usr/bin/Xorg :0 -nr -verb
 2928 razb       20   0  981M  144M 27304 S  0.0  7.4 53:20.45 /usr/lib64/firefox-3.5.4/f
 2328 razb       20   0  805M 22056 15328 S  0.0  1.1  0:41.31 nautilus
 2494 razb       20   0  473M 15252 10204 S  0.0  0.8  0:20.46 /usr/libexec/wnck-applet -
 3123 razb       20   0  387M 13712  9668 S  0.0  0.7  0:25.37 metacity --replace
 7748 razb       20   0  513M 32180 19096 S  0.0  1.6  1:01.62 konsole
    1 root       20   0  4080   932   668 S  0.0  0.0  0:00.39 /sbin/init
  184 root       16  -4 10464   680   312 S  0.0  0.0  0:00.16 /sbin/udevd -d
 1366 root       20   0  250M  2048   884 S  0.0  0.1  0:00.00 rsyslogd -c 3
 1368 root       20   0  250M  2048   884 S  0.0  0.1  0:00.13 rsyslogd -c 3
 1369 root       20   0  250M  2048   884 S  0.0  0.1  0:00.01 rsyslogd -c 3
 1389 rpc        20   0 18788   824   600 S  0.0  0.0  0:00.11 rpcbind
 1403 dbus       20   0 22068  1820   776 S  0.0  0.1  0:32.67 dbus-daemon --system
 1412 avahi      20   0 25512  1484  1228 S  0.0  0.1  0:00.09 avahi-daemon: running [raz
 1413 avahi      20   0 25388   352   180 S  0.0  0.0  0:00.00 avahi-daemon: chroot helpe
 1422 root       20   0  164M  3916  2824 S  0.0  0.2  0:01.78 cupsd -C /etc/cups/cupsd.c
 1443 root       20   0  3916   652   528 S  0.0  0.0  0:00.82 /usr/sbin/acpid
 1451 haldaemo   20   0 30256  4972  3872 S  0.0  0.2  0:17.99 hald
 1454 root       20   0  995M  2696  1628 S  0.0  0.1  0:00.26 /usr/sbin/console-kit-daem
 1456 root       20   0  995M  2696  1628 S  0.0  0.1  0:00.00 /usr/sbin/console-kit-daem
F1Help  F2Setup F3SearchF4InvertF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

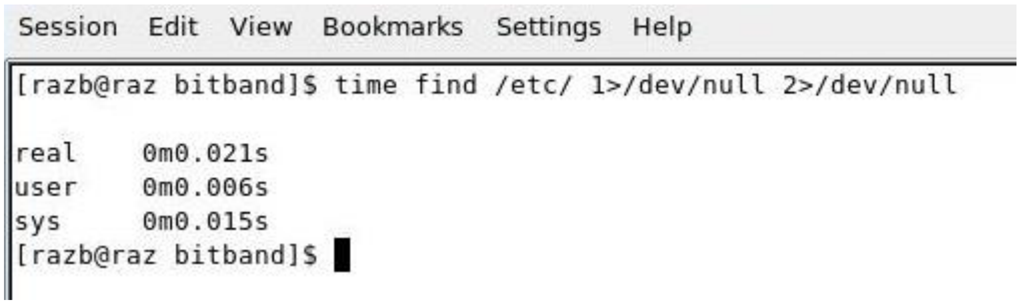
13.1.4 Latencytop

Latencytop is analysis tool aimed to present the user where the latencies are.

```
Cause                                   Maximum     Percentage
Cause                             Maximum     Percentage
Waiting for event (select)               4.6 msec        63.2  %
Waiting for a process to die              1.4 msec         0.7 %
Waiting for TTY data                    0.7 msec        21.3 %
Waiting for TTY input                   0.3 msec        13.9 %
Writing data to TTY                     0.1 msec        0.7 %


Process find (25871)              Total:   2.5 msec
Writing data to TTY                     0.1 msec       100.0 %






 jbd2/sda2-8  Xorg  unity-greeter  sshd  kworker/0:2  kworker/1:1  latencytop
```

13.1.5 Time

The time command argument is another program. Time measures the amount of time the program spent in kernel space (sys) and user space (user). In the bellow find command was running for 21ms, 15ms of it was in kernel while 6ms were spent in user space.



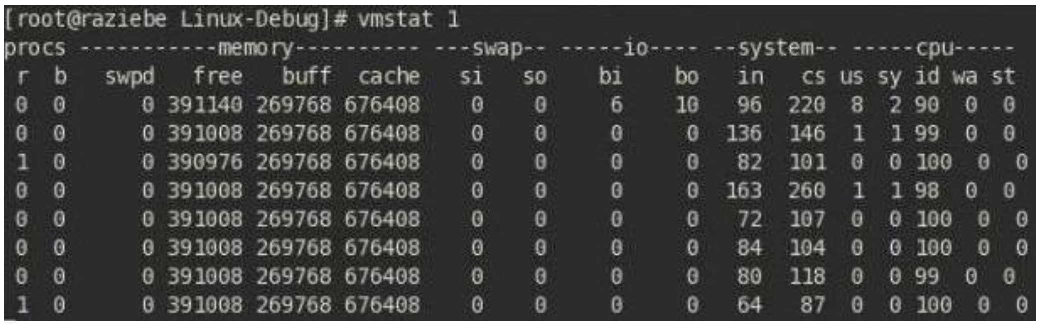
```
[razb@raz bitband]$ time find /etc/ 1>/dev/null 2>/dev/null

real    0m0.021s
user    0m0.006s
sys     0m0.015s
[razb@raz bitband]$ 
```

13.1.6 vmstat

Vmstat is a program that monitors the state of the following resources:

free	free memory in killbytes
buff	amount of buffered memory used by the kernel (for IO operations)
cache	amount of kernel memory
bi	volume of input from the storage layer (in Kilobytes)
bo	Volume if output [1] from the storage layer (in Kilobytes)
swap	amount of swap space[2]
in	number of Interrupts

cs	number of context switches
us	percentage of time processor spent in user space
sy	percentage of time processor spent in kernel space
id	percentage of time processor spent in idle
wa	percentage of time the kernel spent waiting for IO to complete
st	percentage of time stoled from the hosting operating system in favor of guest operating system.



```
[root@raziebe Linux-Debug]# vmstat 1
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 391140 269768 676408    0    0     6    10   96  220  8  2 90  0  0
 0  0      0 391008 269768 676408    0    0     0     0  136  146  1  1 99  0  0
 1  0      0 390976 269768 676408    0    0     0     0   82  101  0  0 100  0  0
 0  0      0 391008 269768 676408    0    0     0     0  163  260  1  1 98  0  0
 0  0      0 391008 269768 676408    0    0     0     0   72  107  0  0 100  0  0
 0  0      0 391008 269768 676408    0    0     0     0   84  104  0  0 100  0  0
 0  0      0 391008 269768 676408    0    0     0     0   80  118  0  0 99  0  0
 1  0      0 391008 269768 676408    0    0     0     0   64   87  0  0 100  0  0
```

The percentage is calculated by the interval time of vmstat. The command "vmstat 1" means 1 second.

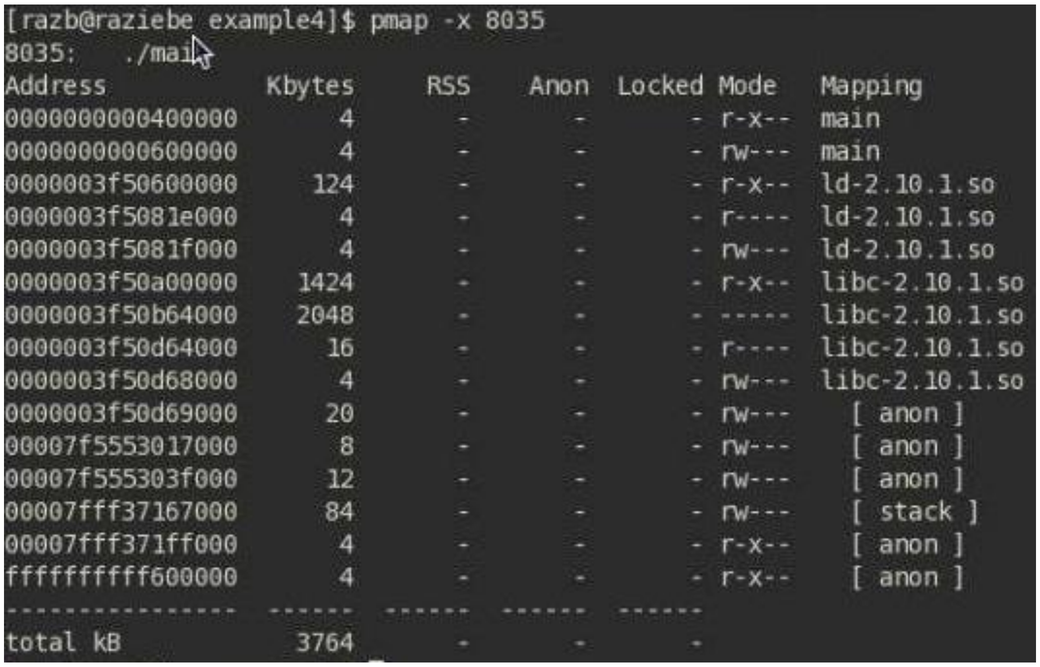
[1] There are times when there is an IO activity but you can't tell who initiated the IO. This IO is likely to be page-out and page-in activity. Paging activity is found in /proc/vmstat. For example:

```
$ grep pgout /proc/vmstat
```

[2] To control swap use swapon/swapoff.

13.1.7 pmap

Pmap – the Process map presents the process memory areas.

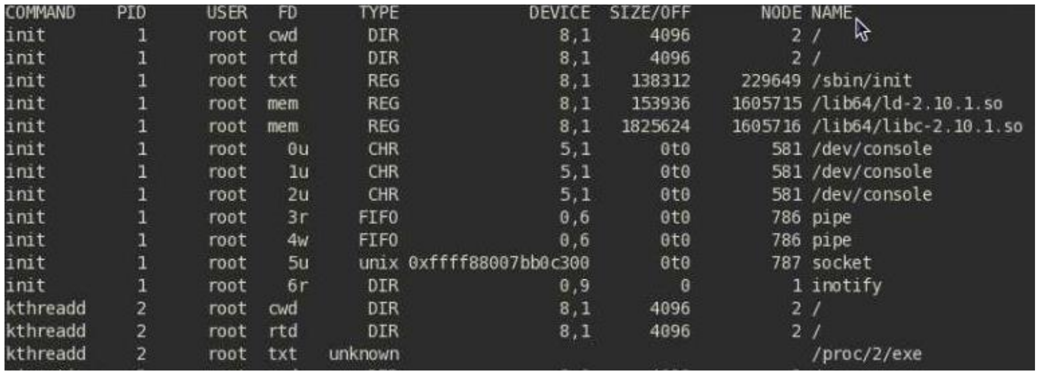


```
[razb@raziebe example4]$ pmap -x 8035
8035:   ./main
Address           Kbytes     RSS    Anon  Locked Mode   Mapping
0000000000400000       4       -       -       - r-x--  main
0000000000600000       4       -       -       - rw---  main
0000003f50600000     124       -       -       - r-x--  ld-2.10.1.so
0000003f5081e000       4       -       -       - r----  ld-2.10.1.so
0000003f5081f000       4       -       -       - rw---  ld-2.10.1.so
0000003f50a00000    1424       -       -       - r-x--  libc-2.10.1.so
0000003f50b64000    2048       -       -       - -----  libc-2.10.1.so
0000003f50d64000      16       -       -       - r----  libc-2.10.1.so
0000003f50d68000       4       -       -       - rw---  libc-2.10.1.so
0000003f50d69000      20       -       -       - rw---    [ anon ]
00007f5553017000       8       -       -       - rw---    [ anon ]
00007f555303f000      12       -       -       - rw---    [ anon ]
00007fff37167000      84       -       -       - rw---    [ stack ]
00007fff371ff000       4       -       -       - r-x--    [ anon ]
ffffffffff600000       4       -       -       - r-x--    [ anon ]
----------------  ------  ------  ------  ------
total kB            3764       -       -       -
```

13.1.8 lsof

lsof list open file descriptors. A file descriptor may be a socket, a pipe, a storage file, IO file and so on.

```
$ lsof
```



```
COMMAND     PID      USER   FD      TYPE             DEVICE  SIZE/OFF       NODE NAME
init          1      root  cwd       DIR                8,1      4096          2 /
init          1      root  rtd       DIR                8,1      4096          2 /
init          1      root  txt       REG                8,1    138312     229649 /sbin/init
init          1      root  mem       REG                8,1    153936    1605715 /lib64/ld-2.10.1.so
init          1      root  mem       REG                8,1   1825624    1605716 /lib64/libc-2.10.1.so
init          1      root    0u      CHR                5,1       0t0        581 /dev/console
init          1      root    1u      CHR                5,1       0t0        581 /dev/console
init          1      root    2u      CHR                5,1       0t0        581 /dev/console
init          1      root    3r     FIFO                0,6       0t0        786 pipe
init          1      root    4w     FIFO                0,6       0t0        786 pipe
init          1      root    5u     unix 0xffff88007bb0c300       0t0        787 socket
init          1      root    6r      DIR                0,9         0          1 inotify
kthreadd      2      root  cwd       DIR                8,1      4096          2 /
kthreadd      2      root  rtd       DIR                8,1      4096          2 /
kthreadd      2      root  txt   unknown                                         /proc/2/exe
```

13.1.9 System Lockup

System Lockup is if any CPU in the computer does not execute the local timer interrupt for some time. For this Linux kernel designers created a watchdog called NMI watchdog. NMI stands for Non Mask-able Interrupt. This watchdog is executed in NMI interrupt context and as such it cannot be masked. So even if the kernel is deadlock when interrupts are locked, the nmi watchdog will be launched and dump an oops and kill the current process.

To use the nmi watchdog boot the kernel with nmi_watchdog=1 or nmi_watchdog=2.

13.1.10 Overly stressed operating system

There are times when your system is so loaded that and you cannot do anything to get information from it. Even when running with gdb does not help.

Process Affinity

If you have a dual core machine, or hyper-threading, simply affine your task and all interrupts to one core and leave the other processor idle as much as possible. This sometimes works.

The Preempt RT trick

This trick is possible mostly on preempt RT kernels. It might work in regular kernels. Idea is simple , boost both secure shell and your bash shell priority to maximum.

13.2 Kernel configuration

Kernel configuration usually means the configuration that the kernel was built with.

A programmer should aim his program to run correctly in each configuration or require the kernel configuration as part of the software delivery.

13.2.1 Ticks - HZ

A kernel may be configured to tick at 100HZ, 250HZ, 300HZ, 1000HZ.

13.2.2 A Pre-emption model

There are four main kernel preemption models:

1. Server – no Preemption
2. Low Latency desktop
3. Full preemption model.
4. Real time preemption

13.3 Errors Simulation

Program failures happen in various system conditions; for example, when system is low on memory or storage. We can never be too careful when testing your program in various conditions.

The least we expect from the software is to report what sort of problem it encounters.

13.3.1 Simulate Storage Failures

FULL FILE SYSTEM

When a Linux file system is full the system is buggy and unstable; and adding to that it is not easy to find this problem. A good practice is to have a demonized process running checking if system is reaching its fullness.

Write a program that fills the root file system to check your daemon. Use the df command.

IO ERRORS

It is good practice to check for errors after reading or writing to a file.   But how do we test if the program behaves the way we designed it. We can add code that simulates IO errors in the program. But this means we add a wrapper for each IO operation made.

There are other ways to simulate IO Errors. For example we can use scsi_debug driver to simulate IO errors. First we need to load the driver with some additional parameters:

```
$ sudo modprobe scsi_debug  dev_size_mb=1 delay=5 every_nth=13
```

The modprobe command loads a driver, that creates a virtual scsi disk of size of 1MB, each IO operations costs 5ms, and every 13-th operation is timed out – meaning it fails.

Now, we need to know which disk was loaded. For that enter dmesg.

```
$ dmesg
```

A log file will print to screen, at the bottom you are likely to see the new disk.

```
[11818.241587]  scsi4 : scsi_debug, version 1.82 [20100324],  dev_size_mb=10, opts=0x0
[11818.242309]  scsi 4:0:0:0: Direct-Access    Linux    scsi_debug       0004  PQ: 0 ANSI: 5
[11818.242452]  sd 4:0:0:0: Attached scsi generic sg2 type 0
[11818.361090]  sd 4:0:0:0: [sdb] 2048  512-byte logical blocks: (1.04 MB/1.0 MiB)
[11818.481051]  sd 4:0:0:0: [sdb] Write Protect is off
[11818.481057]  sd 4:0:0:0: [sdb] Mode Sense: 73 00 10 08
[11818.720969]  sd 4:0:0:0: [sdb] Write cache: enabled, read cache: enabled, supports DPO and
FUA
[11819.560728]   sdb: unknown partition table
[11820.040559]  sd 4:0:0:0: [sdb] Attached SCSI disk
```

So /dev/sdb is the new disk. Let us format it.

```
$ mkfs.ext2 /dev/sdb
```

Now we mount it.

```
$ sudo mount /dev/sdb /mnt
```

The mount operation will take longer than expected, because we set the disk to report a timeout error every 13-th iop. Also the more we increase the delay the less responsive disk we get.

Now we can check the speed of the disk by dd'ing a file. For example:

```
dd if=/dev/random of=/mnt/TEST count=10 ibs=1 obs=10
```

Write your own program and experiment with scsi_debug. If your program look for a file in /opt/foo make /opt/foo a symbolic link to /mnt/foo , this way you will not change the program code.

13.3.2 Simulate Memory stress

What happen when malloc fails? Will the program crash? A fast way to check that is boot the kernel with a parameter that decrease the amount of RAM the operating sees. In the example bellow I restricted the operating system to 200 MB RAM.

```
title CentOS (2.6.18-8.el5)
        root (hd0,1)
        kernel /boot/vmlinuz-2.6.18-8.el5 ro root=LABEL=/1          mem=200m
        initrd /boot/initrd-2.6.18-8.el5.img
```

13.3.3 Simulate number of processors

Your program may be executed in various types of machines. Some may have a single processor and some may have multiple processors. So, in the case you do not have at your possession a single core machine, you can remove a processor from the operating system.

In the example bellow, I removed from a dual core machine one processor.

```
$ echo 0 > /sys/devices/system/cpu/cpu1/online
```

13.3.4 Processors isolation

There are times where we want the program would be running in an isolated processor but still have the other remaining processors available for other system operations. This is possible by using the boot command line isocpus and assign the program to this processor directly.

Another possibility is to use cpu groups. This is possible through a series of mounting operations. This advantages of cgroups is that it possible to change the configuration in real time while is isocpus it is not possible.

14. Miscellaneous

14.1 Binutils

Binutils is a suite of commands that come with the compiler. Here are some.

14.1.1 Objdump

Objdump displays information from object file. Objump comes in handy for interpreting assembler code interlaced with c code. For example:

```
$ objdump –Sl a.out
```

14.1.2 Readelf

Readelf is a command that presents a binary format. It is useful when examining a binary sections and there are times that it replaces objdump.

14.1.3 Strip

Strip is a command that remove debug information from an object file. it quite useful when debugging with remote gdb.

14.1.4 Nm

Nm Lists symbols of an object file.

14.1.5 Ldd

Ldd presents a list of libraries that are loaded when the binary is executed.

14.2 Overriding Functions

The LD_PRELOAD environment variable is used to load dynamic shared objects prior to other libraries. The common use for it is to override functions, see how Duma overrides malloc and free of libc.

You received a third-party library. You use it but there are times that it crashes with SIGPIPE. You are willing to fail but you cannot crash.

Fix it. Library is called liberror.so and main.c needs to link with it.

14.3 Memory barriers

Processors might reorder store and loads commands in real time due to performance considerations.

user code		actual flow	
thread	**device/thread**	**thread**	**device/thread**
a = 2 ...		a = 2...	
a = 3		b = 4	
b =4	c = b	a = 3	c = 4
	d = a		d = 2

Memory barriers fix this. Read or Write memory barriers.

thread	**thread/device**	**flow**	
a = 2 ...		a = 2...	
a = 3		a = 3	
mb()			
b =4	c = b	b = 4	d = 3
	d = a		c = 4

Why is it happening?

It is happening because CPU caches are divided into independent banks. Each bank may access the RAM in parallel to the other bank and each bank is responsible to fetch and store addresses. In the example bellow, bank 0 is responsibility is even number **cache lines**, while bank 1 responsibly is odd number **cache lines**.

Note!

The "way 0" and "way 1" bellow refer to the cache associativity level. Associativity level means: "to how many places an addresses can me mapped in the cache". In the bellow it is 2.

For example,

Example for 2-way cache with 256 bytes cache line

Address	Way 0	Way 1
0x00	0xbc012000	
0x01	0xbc012100	
0x02	0xbc012200	
0x03	0xbc012300	0xbc455300
0x04	0xbc012400	
0x05	0xbc012500	0x12019500

If a program executing in cpu 0 accesses two consequent lines, one is even and the other is odd. For example, if program accesses addresses 0xbc012200 and 0xbc012300 , which maps to lines 0x03 and 0x02, it is possible that line 0x03 may populate before line 0x02. So if cpu 1 looks at the cache he sees the data in cache 0x03 before 0x02.

As the reader probably understood, it is very difficult to understand how and when the processor re-ordered code. So, the author suggests using the litmus software package to create scenarios where memory reordering might take place. The litmus package can be found at the kernel course or in http://diy.inria.fr/doc/litmus.html.

Litmus used the ocaml compiler. Caml is a programming language that was initially created to prove automation theorems, compiler and interpreters. In our case, Caml is used to validate whether memory re-reordering took place. Please download ocaml from http://caml.inria.fr/download.en.html and perform the following actions:

```
$ tar xvf ocaml-<version number>.tar.gz

$ cd ocaml-<version number>/

$ make

$ sudo make install
```

Now go to the course/mem_barrier/litmus directory and make:

```
$ make all

$ sudo make install
```

If all goes well, litmus is installed. Now, let us demonstrate a simple memory-reodering test. This test is called class.litmus and can be downloaded from the litmus website.

```
$ litmus class.litmus
```

Results will be dumped to screen. Look for the Histogram. In my machine results are:

Executing on Intel core i5-2320 3Ghz.

```
Test classic Allowed
Histogram (3 states)
2     :>0:EAX=0; 1:EAX=0;
499999:>0:EAX=1; 1:EAX=0;
499999:>0:EAX=0; 1:EAX=1;
Ok
```

The positive number “2” means that memory re-ordering took place, 2 cycles out of a million we re-ordered.

Now, let me explain the test. classic.litmus is written in caml language.

```
MOV [y],$1
```

Means y = 1

and

```
MOV EAX,[x]
```

Means register EAX = x; same logic applies to the second block but with y. In all test initially X = 0; Y= 0;

Cpu 0	Cpu 1	Direction
Y =1	X =1	Write
R1 = X	R2 = Y	Read

Under **sequential consistency memory model**, at least one WRITE must be made before the any of the READ.

R1 = 1: R2=1 → Cpu0 and Cpu1 ended concurrently.

R1 = 0: R2 = 1 → Cpu0 finished first.

R1 = 1: R2= 0 → Cpu1 finished first.

R1 =0: R2 =0 → Impossible under sequential consistency memory model.

Yet, as depicted above, I got 2 out of a million iterations. How can that be?

Reason is that the write command was made AFTER the read command because of memory re-ordering took place.

Re-write the above example in c. show that this can happen in c code as well.

14.4 GCC optimization bugs

GCC optimization is a technique used by the compiler to boost the program performance. One of the optimizations GCC perform is to reduce the code size. Yet this not comes without a cost. One of the problems is that the produced binary code may NOT as the writer intended. Understanding these differences may save the programmer un-pleasant bugs.

14.4.1 Optimization barrier

The compiler may decide to remove some command as it believes the code is executed in serial manner. Here is an example:

```
int foo(
	int *z,
	int *y,
	int *x,
	int *lock)
{

	*lock = 0x1;
	*y = 0x5;
	*x = 0x8;
	*z = 0x12;
	*lock = 0x0;
	if (*z == *y)
		*y = 0xdd;
	else
		*y = 0xff;
}
```

Now say the programmer assumes that some other execution context checks if whether the lock is ever 1 for some reason he would find this forever false because lock is never 1 when code is optimized with O2.

Compile this code with –O2 and then with –O0 and use objdump –Sl to examine the output.

Use sync_synchronize() to solve it.

14.4.2 Unpredictable

The bellow code has a bug. What do you think this routine will return when x is zero?
Compile this code with and without optimization and figure out yourself.

```
int func(int x)
{
        int a;
        if (x)
                a = 42;
        return a;
}
```

14.4.3 Warn unused

There are functions that when used their return value must be checked. We can ask the compiler to generate a warning if the code somehow ignores it.

```
__attribute__ ((warn_unused_result)) int __bigger(int x, int y)
{ return x>y;  }
...
__bigger(1,2)
....
```

The compiler would generate the following error.

```
warning: ignoring return value of max, declared with attribute warn_unused_result [-Wunused-
result]
```

14.4.4 GCC's sections

The default behavior of the compiler is to place the global variables in the bss section. However; there are times where you need to place code or global variables in different sections. For this use the section attribute.

```
int foo __attribute__ ((section ("bar")))
```

The above puts the function `foo' in the `bar' section.

To examine the sections, use "readelf" command.

```
 struct __data__ {
        unsigned int n;
};
```

```
void *Run(void *data)
{
        int i, j;
        struct __data__ *com = (struct __data__ *)data;


        for (j = 0; j < 20000; j++) {
                for (i = 0; i < 100000; i++) {
                        com->n++;
                }
        }

        return 0;
}

struct __data__ com1;
struct __data__ com2;

int main(int argc, char *argv[])
{
        int i;
        int j;
        pthread_t t;

        if (pthread_create(&t, NULL, Run, (void *)&com1) < 0) {
                perror("pthread create");
                return 0;
        }
        Run(&com2);
}
```

The above code can run twice as fast by using sections.

14.4.5 GCC c Labels

From time to time programmer use the goto label in the code. Gcc 4.x introduced lables as values. To assign a label to a value use &&.

```
void *ptr;
foo1:
....
foo2:
...
If <some condition>
   ptr = &&foo1;
else
   ptr = &&foo2;
goto ptr;
```

Create a mathematical function that uses the following unary operators: shift left,shift right, square root, power 2.

Implement only by labels.

14.5 Code analysis with Git

Git is revision control system like svn or cvs. Git has some useful debug features.

14.5.1 Git bisect

Git bisect is a built-in regression test. Many times a program fails to do what she was doing some time ago because of a bad commit. So programmers usually go back to the last workable commit and start climbing up searching for the commit that fails the program.

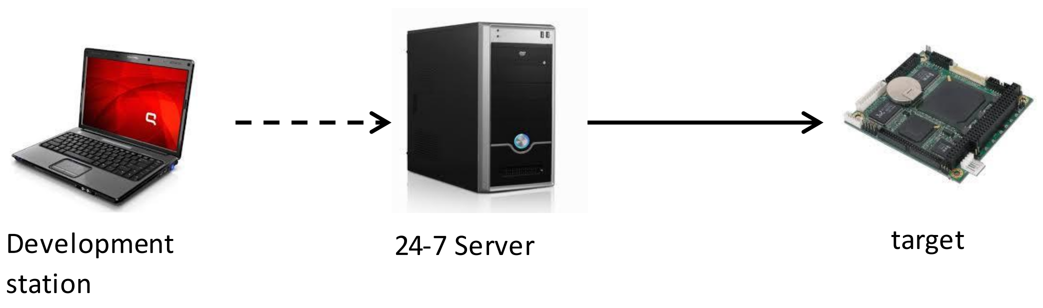
Git bisect is a command that helps with the process of finding this bad commit(s). It does so by setting two points, A and B in the commit history. Point A is supposed to be the commit where program works without a failure and point B is the point where the program is known to fail. Git bisect provides a process of logarithmic search in the commit history

14.5.2 Git blame

Git blame is built-in git command that provides a commit history in the line level. It is useful where several team members work closely on the same piece of code.

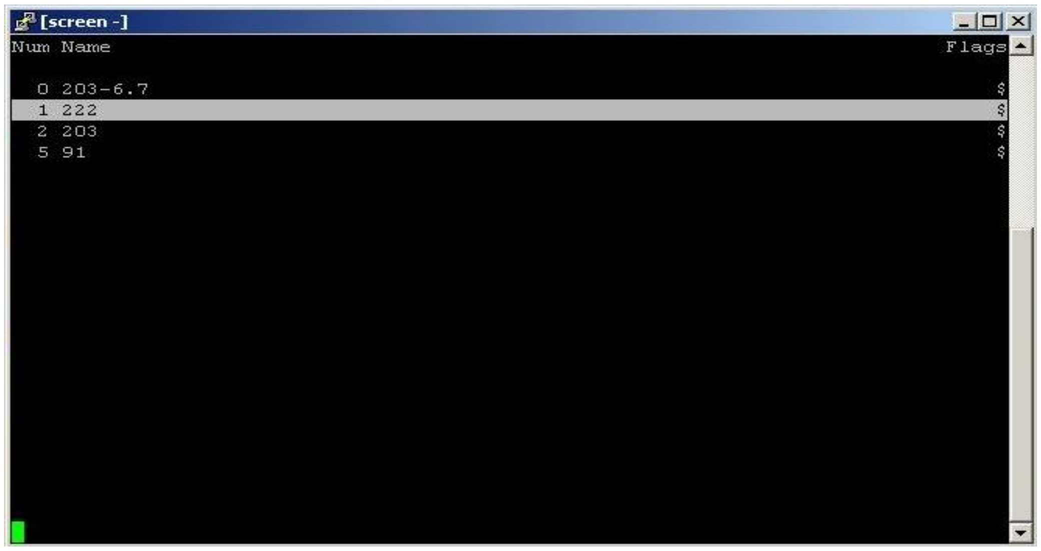
14.6 Screen

Screen is a textual window manager. Though it is not directly related to debugging, we use screen because it enables us to leave open login windows for days. This way we can execute commands interactively and disconnect from the window. It is sometimes important because interactive program.



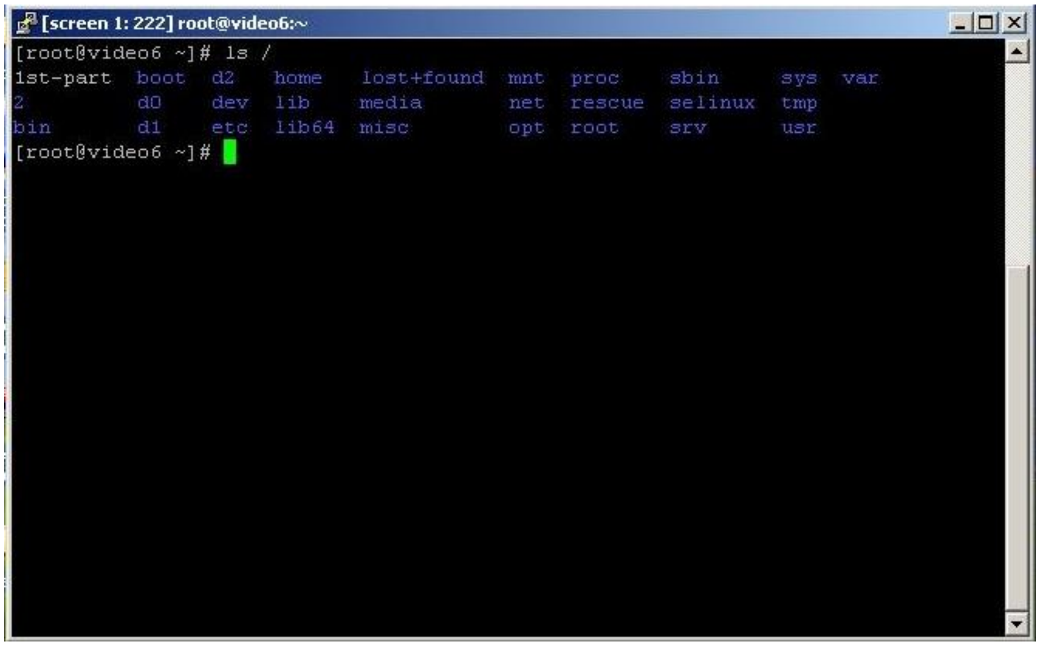
Developer connect (through ssh) to the server. He opens a screen session. At the end of the day he disconnects from the screen. At the morning he connects back to the server are attaches to screen windows manager as if he never disconnected.

The bellow is how screen looks like. Each line presents a different session. I am choosing the session named 222.



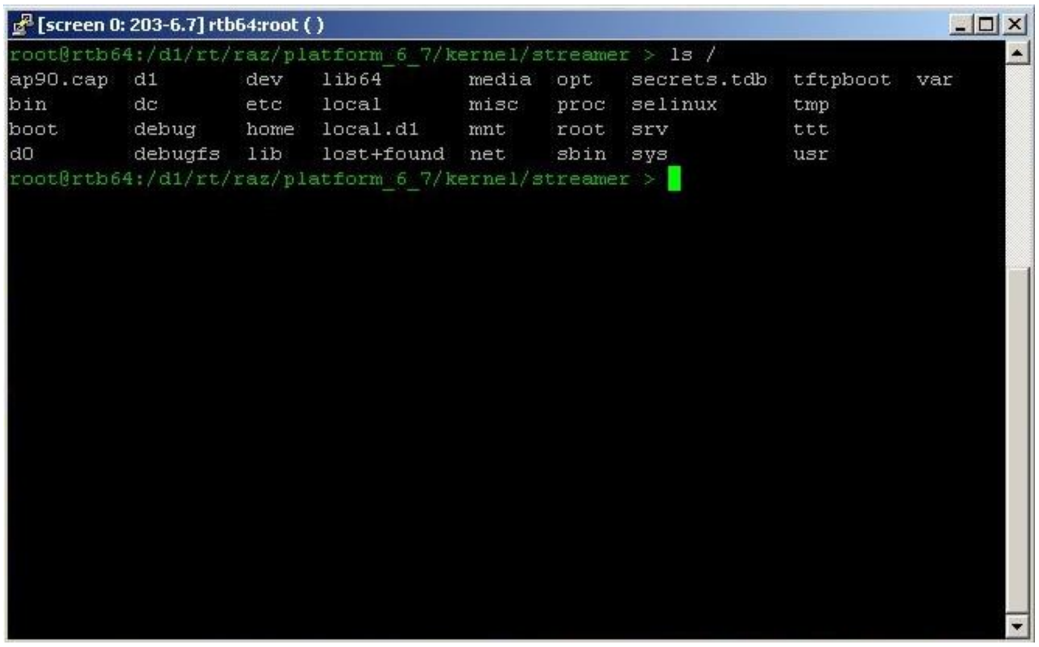
```
[screen -]
Num Name                                                              Flags

  0 203-6.7                                                               $
  1 222                                                                   $
  2 203                                                                   $
  5 91                                                                    $
```

And this is how session 222 looks like.



```
[screen 1: 222] root@video6:~
[root@video6 ~]# ls /
1st-part  boot  d2   home   lost+found  mnt  proc    sbin     sys  var
2         d0    dev  lib    media       net  rescue  selinux  tmp
bin       d1    etc  lib64  misc        opt  root    srv      usr
[root@video6 ~]#
```

Now I switched to session named 203-7.



```
root@rtb64:/d1/rt/raz/platform_6_7/kernel/streamer > ls /
ap90.cap  d1       dev   lib64       media  opt   secrets.tdb  tftpboot  var
bin       dc       etc   local       misc   proc  selinux      tmp
boot      debug    home  local.d1    mnt    root  srv          ttt
d0        debugfs  lib   lost+found  net    sbin  sys          usr
root@rtb64:/d1/rt/raz/platform_6_7/kernel/streamer >
```

You can detach from the screen with ctrl +d.

You can climb up to see the commands that left screen with ctrl +A[.

15. An Embedded Linux Project

This chapter describes how to make a raspberry PI a surveillance camera with very little coding. This camera uses IoT mqtt to connect to a smart phone and pass information.

15.1 Hardware

This is a Raspberry PI computer. It runs Debian Linux. It is an ARM processor. Raspberry consumes very little power.

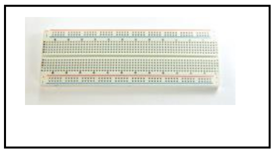
A micro SD. This is the storage device.

Flash card reader. We use it to flush the Debian operating system.

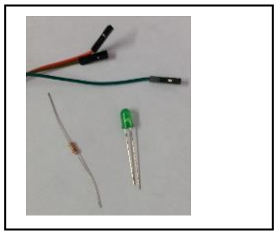
A Wifi dongle. This dongle is used connect to the internet or the smear phone.

USB camera.

Raspberry PI power cord



A Matrix. We use it to create a simple electrical circuit.



Some matrix compatible wires and two leds.

Rs232 USB converter. This connects to the laptop or development PC through an extender.

Usb Extender. Connect it to the Rs232 usb converter

Wifi router to connect the PI.

Smart phone.

Development laptop or PC.

15.2 Host Requirements

The development workstation should be running Ubuntu or any other Linux based desktop. It also needs to a Raspberry PI toolchain, some utilities like wget, dd and others. And internet.

15.3 PI's Software

The PI would boot a Debian image file. Then will download the Motion utility from the internet into the target. Would would also need a console editor, like vi or nano. Last package would be the Mqtt package.

15.4 Preparation

Download the image file and flash sd card with card reader. Use the dd command:

```
$ sudo dd if=pi.img of=/dev/sdb bs=1M
```

Connect Rs232 sub converter to the laptop and launch minicom. Then power up the Raspberry PI and boot it. You should be able to see the boot process takes place. The boot process ends when the following appears in minicom.

```
$login:
```

The user is pi the password is raspberry.

Turn of the PI. Connect the wifi dongle and boot again. Once you login again ping google to check for internet access. Once there is an internet access install ssh.

```
$ sudo apt-get install sshd
```

Check for ip by using the command ifconfig.

```
$ ifconfig
```

Now login from your laptop to the Raspberry PI with ssh.

15.4.1 Camera setup

Download motion.

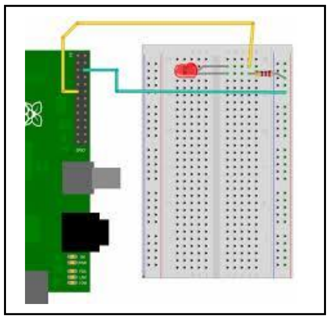
```
$ sudo apt-get install motion
```

Connect the usb camera. Run motion and see that camera is turned on.

Configure motion. Fix the frame rate and if needed the video device name.

15.4.2 Gpio

Connect the wires and the leds. Check if the leds are lighting up and powering off.



15.4.3 Mqtt

Download mqtt client to target machine. Download to your smart phone mqtt . Check for connectivity.

16. Bibliography

Building Embedded Linux SYSTEMS. 2nd Edition August 2008 . Karim Yaghmour, Jon Masters, Gilad Ben Yosef, Philppe Gerum. ISBN 978-0-596-52968-0. Oreily Media

Understanding the Linux Kernel. 3rd Edition. Daniel P.Bovet and Marco Cesati. 2006 Oreilly Media

Linux Kernel Development. Second edition. Robert Love. ISBN 0-672-32720-1. January 2005. Prentice Hall

The definitive Guide to GCC. Second Edition. William Von Hagen. ISBN-13 978-1-59059-585-5. APRESS.

Linux Debugging and Performance - Tuning Tips and Techniques. Steve Best. ISBN 0-13-149247-0 . October 2005. Prentice Hall