# introduction to linux kernel and application profiling

## goal

The goal of this Lab is to allow you to use the Arm Development Studio Streamline tool to profile the execution of code running on the Raspberry Pi 3.

## prerequisites

To follow this Lab, you need:

1. Raspberry Pi 3 board full;
2. Micro USB cable;
3. 8 GB Micro SD card;
4. USB-to-Serial debug module for Raspberry Pi 3 or USB to TTL adapter;
5. A PC with Ubuntu Desktop 14.04 LTS, or a virtual machine hosting Ubuntu Desktop 14.04 LTS;
6. A Micro SD card reader attached to the PC/virtual machine;
7. An Ethernet cable;
8. The Arm Development Studio tool installed on your development host (Will cover this in the lab);
9. (**Optional**) Micro HDMI cable.

## workplace setup

Assuming you completed all the previous labs, move to the directory ***raspberryPi3*** and prepare the build environment:

cd ~/raspberryPi3

source sources/poky/oe-init-build-env rpi-build

You now have the system ready for building the embedded Linux distribution for the Raspberry Pi 3. We now need to:

* Customize the configuration of the Linux kernel to activate the functionalities needed to run code profiling, which is measuring the execution time of the code on the Raspberry Pi 3 target;
* Prepare the Yocto recipe to deploy Streamline-specific kernel modules/middleware to the Raspberry Pi 3.

## Configuring the kernel

To configure the Linux kernel, we have to modify, using the kernel configuration utility, the Linux configuration file (.config). For this purpose, we proceed as follows.

Run the following command:

bitbake linux-raspberrypi -c devshell

This command will tell the Yocto build system to prepare the Linux kernel compilation (***bitbake linux-raspberrypi***) and to open a development shell (option ***-c devshell***) with the build system configured for the selected recipe (***linux-raspberrypi***). From this development shell, we can run the Linux kernel configuration utility as follows:

make menuconfig

Customize the Linux kernel configuration as follows (in the following, the asterisk tells the option shall be activated):

General setup --->

[\*] Embedded system

Kernel Performance Events and Counter --->

[\*] Kernel Performance Events and Counter

[\*] Profiling support

Kernel features --->

[\*] Enable hardware performance counter support for perf events

Kernel hacking --->

-\*- Kernel debugging

[\*] Tracers --->

[\*] Kernel Function Tracer

[\*] Kernel Function Tracer Graph

[\*] Interrupts-off Latency Tracer

[\*] Preemption-off Latency Tracer

[\*] Scheduling Latency Tracer

[\*] Trace syscalls

-\*- Create a snapshot trace buffer

-\*- Allow snapshot to swap per CPU

[\*] enable/disable function tracing dynamically

[\*] Kernel function profiler

When done with this configuration, you can exit from the Linux configuration utility and save the configuration to the .config file. The current configuration is valid only for the current instance of the ***bitbake*** command. If you close the development shell, thus terminating the current instance of the ***bitbake*** command, the next instance of the ***bitbake*** command will bring the configuration back to the default one. Therefore, we have to make the current configuration the new default one. For this purpose, we have to copy the .config file to the default configuration file in the ***linux-raspberrypi*** recipe as follows:

cd /home/user/raspberryPi3/rpi-build/tmp/work/raspberrypi3-poky-linux-gnueabi/linux-raspberrypi/1\_4.1.21+gitAUTOINC+ff45bc0e89-r0/linux-raspberrypi3-standard-build

This directory will be printed to the console after closing the kernel configuration utility; tailor this to the directory your system prints.

cp .config ~/raspberryPi3/sources/meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi/defconfig

With this command, we make the modifications to the Linux kernel configuration permanent.

## Preparing the recipes

In order to profile the execution of the code on the Raspberry Pi 3, we will exploit the Streamline tool, part of the Arm Development Studio suite.

To install Arm Development Studio trial version, go to https://developer.arm.com/tools-and-software/embedded/arm-development-studio.

Select "Try for free" and follow the on-page instructions for downloading and installing it.

Streamline makes use of three cooperating elements to execute the profiling:

* ***gator.ko***, it is a loadable kernel module that shall be installed on the Raspberry Pi 3 to provide user-level access to the performance counter of the Arm architecture;
* ***gatord***, it is a service that shall be installed on the Raspberry Pi 3 that interacts with gator.ko to collect the performance counter values and to deliver them through the Ethernet network to the development host;
* ***streamline***, it is an application running on the development host to configure the operations of ***gator.ko/gatord*** and to display the gathered execution traces.

To integrate ***gator.ko/gatord*** in the Linux system for the Raspberry Pi 3, we have to prepare a novel Yocto recipe to instruct the build system: where to find the needed source code, how to compile it, and how to deploy it in the root file system.

For this purpose, create the directory as follows:

mkdir ~/raspberryPi3/sources/meta-raspberrypi/recipes-kernel/gator

mkdir ~/raspberryPi3/sources/meta-raspberrypi/recipes-kernel/gator/files

In ***~/raspberryPi3/sources/meta-raspberrypi/recipes-kernel/gator***, put a file named ***gator\_1.0.bb*** with the following content:

|  |  |
| --- | --- |
| 0:  1:  2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  34:  35: | DESCRIPTION = "ARM Development Studio Gator kernel module and daemon"  SECTION = "kernel"  LICENSE = "GPLv2"  LIC\_FILES\_CHKSUM="file://driver/COPYING;md5=b234ee4d69f5fce4486a80fdaf4a4263"  S="${WORKDIR}/git"  BP="${BPN}"  DEPENDS = "linux-raspberrypi"  inherit module  SRCREV = "9cbfb3f6b60664a31bc35745da9c5992d6306352"  SRC\_URI = "git://github.com/ARM-software/gator.git;protocol=https \  file://0001-patch-dameon-Makefile.patch"  INHIBIT\_PACKAGE\_DEBUG\_SPLIT = "1"  INHIBIT\_PACKAGE\_DEBUG\_SPLIT-dev = "1"  INSANE\_SKIP\_${PN}-dev += " ldflags"  INSANE\_SKIP\_${PN} += " ldflags"  do\_compile() {  cd ${S}/driver  ${MAKE} -C ${STAGING\_KERNEL\_DIR} M=`pwd` ARCH=${TARGET\_ARCH} CROSS\_COMPILE=${TARGET\_PREFIX} modules  unset CFLAGS CPPFLAGS CXXFLAGS LDFLAGS MACHINE  cd ${S}/daemon  ${MAKE}  }  do\_install() {  INIT\_DIR=${D}${sysconfdir}/init.d/  install -d ${INIT\_DIR}  install -m 0644 ${S}/driver/gator.ko ${INIT\_DIR}  install -m 0755 ${S}/daemon/gatord ${INIT\_DIR}/gatord  echo "#!/bin/bash\n/etc/init.d/gatord &" > ${INIT\_DIR}/rungator.sh  chmod a+x ${INIT\_DIR}/rungator.sh  }  FILES\_${PN} = "${sysconfdir}/init.d/gator.ko ${sysconfdir}/init.d/gatord ${sysconfdir}/init.d/rungator.sh" |

This file describes the property of the recipe, and in particular:

* Line 4 specifies which is the working directory to be used for downloading/building the code. Upon executing the recipe, Yocto will access the Arm github and download the source code to the defined directory where the gator directory contains the source code for the loadable kernel module, and gatord contains the source code for the service;
* Lines 8-9-11 instruct Yocto to download the source code for gator.ko/gatord from the Arm github so that always the latest distributed source code is used;
* Line 11 specifies the patch file to adapt the gatord Makefile to the Yocto build environment;
* Lines 19-25 specify how the loadable kernel module (gator.ko) and the middleware (gatord) shall be compiled/linked;
* Lines 27-35 instruct Yocto how to deploy the obtained binaries to the target root file system; three new files will be created on the Raspberry Pi 3 root file system: ***/ect/init.d/gator.ko***, ***/etc/init.d/gatord***, ***/etc/init.d/rungator.sh***

In ***~/raspberryPi3/sources/meta-raspberrypi/recipes-kernel/gator/files***, put a file named ***0001-patch-daemon-Makefile.patch*** with the following content:

|  |  |
| --- | --- |
| 0:  1:  2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12: | --- a/daemon/Makefile 2017-01-23 05:44:19.211481780 -0800  +++ b/daemon/Makefile 2017-01-23 05:47:34.805752423 -0800  @@ -8,8 +8,8 @@  # targets run 'make SOFTFLOAT=1 SYSROOT=/path/to/sysroot', see  # streamline/gator/README.md for more details  -CC = $(CROSS\_COMPILE)gcc  -CXX = $(CROSS\_COMPILE)g++  +#CC = $(CROSS\_COMPILE)gcc  +#CXX = $(CROSS\_COMPILE)g++  ifeq ($(SOFTFLOAT),1)  CPPFLAGS += -marm -mthumb-interwork -march=armv4t -mfloat-abi=soft |

The file tells how to modify the gatord ***Makefile*** to use the cross-compiler environment Yocto provides.

The (simplified) Yocto operation flow is as follows:

* Download the source code from the SRC\_URI;
* Apply the patch according to the .patch files provided with the recipe;
* Compile the source code according to the default rules of the do\_compile() section of the recipe;
* Execute the do\_install() section of the recipe, if any.

Once these operations are completed, you have to tell the machine layer configuration that the new driver is needed. For this purpose, edit the file

***raspberryPi3/sources/meta-raspberrypi/conf/machine/raspberrypi3.conf***

and modify it to include:

MACHINE\_EXTRA\_RRECOMMENDS += " kernel-modules wl18xx-conf uim-sysfs bt-firmware gator"

Add the following statement as the last line of the file:

MACHINE\_ESSENTIAL\_EXTRA\_RRECOMMENDS += "kernel-module-gator"

Now you need to ensure that the local.conf file includes gator in the build by adding the line:

IMAGE\_INSTALL\_append += "gator"

Also add:

INSANE\_SKIP\_gator\_forcevariable = " ldflags already-stripped"

This will signal that the gator module is already-stripped and the build process will not reattempt it.

## Building and deploying the new system

You are now ready to build the new system as follows:

bitbake -c clean rpi-basic-image

bitbake rpi-basic-image

After a while, a new Micro SD card image would be available, which you can deploy in the Micro SD as follows (assuming the Micro SD is available to the PC as /dev/sdN). Alternatively, use a program of your preference to flash the image.

First, run the:

sudo fdisk -l

command to determine which device to flash to (plug in and unplug the SD card to determine which device it is). For this example, the SD card is under the name “sdc” (this may be different in your environment). Next, ensure that the device is unmounted. This can be done using the command:

sudo umount /dev/sdc\*

Once this is done, the following command can be used to copy the image across to the SD card (substitute any folder names and device names to ensure they are relevant to your specific environment).

sudo dd bs=1M if=/home/user/raspberryPi3/rpi-build/tmp/deploy/images/raspberrypi3/rpi-basic-image-raspberrypi3.rpi-sdimg of=/dev/sdc

Note that if not done properly, the image being flashed across to the SD card may cause problems when attempting to turn on the board. If this is the case, it may be worth retrying the process again and ensuring that it is done properly, or use a flash program to automate the process.

Also note that this time, the most recently built image should be in the “tmp” folder, not “tmp-glibc”.

## Running the module

After booting the new Linux system, and logging into the Raspberry Pi 3, you can type the following commands:

root@raspberrypi3:/# insmod /etc/init.d/gator.ko

root@raspberrypi3:/# /etc/init.d/rungator.sh

The above commands insert the ***gator.ko*** module into the Linux kernel and run the gatord service.

Finally, we have to set up the IP address of the target by editing the file ***/etc/network/interfaces*** and to modify the configuration of the network device ***eth0*** as follows:

# Wired or wireless interfaces

auto eth0

iface eth0 inet static

address 192.168.1.2

netmask 255.255.255.0

gateway 192.168.1.255

**Make sure that the Raspberry Pi 3 is connected to the development host via Ethernet cable and that the IP address of the Ethernet connection on the development host is 192.168.1.1.**

## Profiling the target

To profile code execution on the Raspberry Pi 3, we have to start Streamline on the development host, but first need to add it to the PATH:

Open the “***/etc/environment***” file:

sudo gedit /etc/environment

Then add the path to Streamline to the “PATH” string. In this case, the path to Streamline is: “/home/user/DS-5\_CE\_v5.29.1/bin” as this is where it was installed.

Save the file and then run:

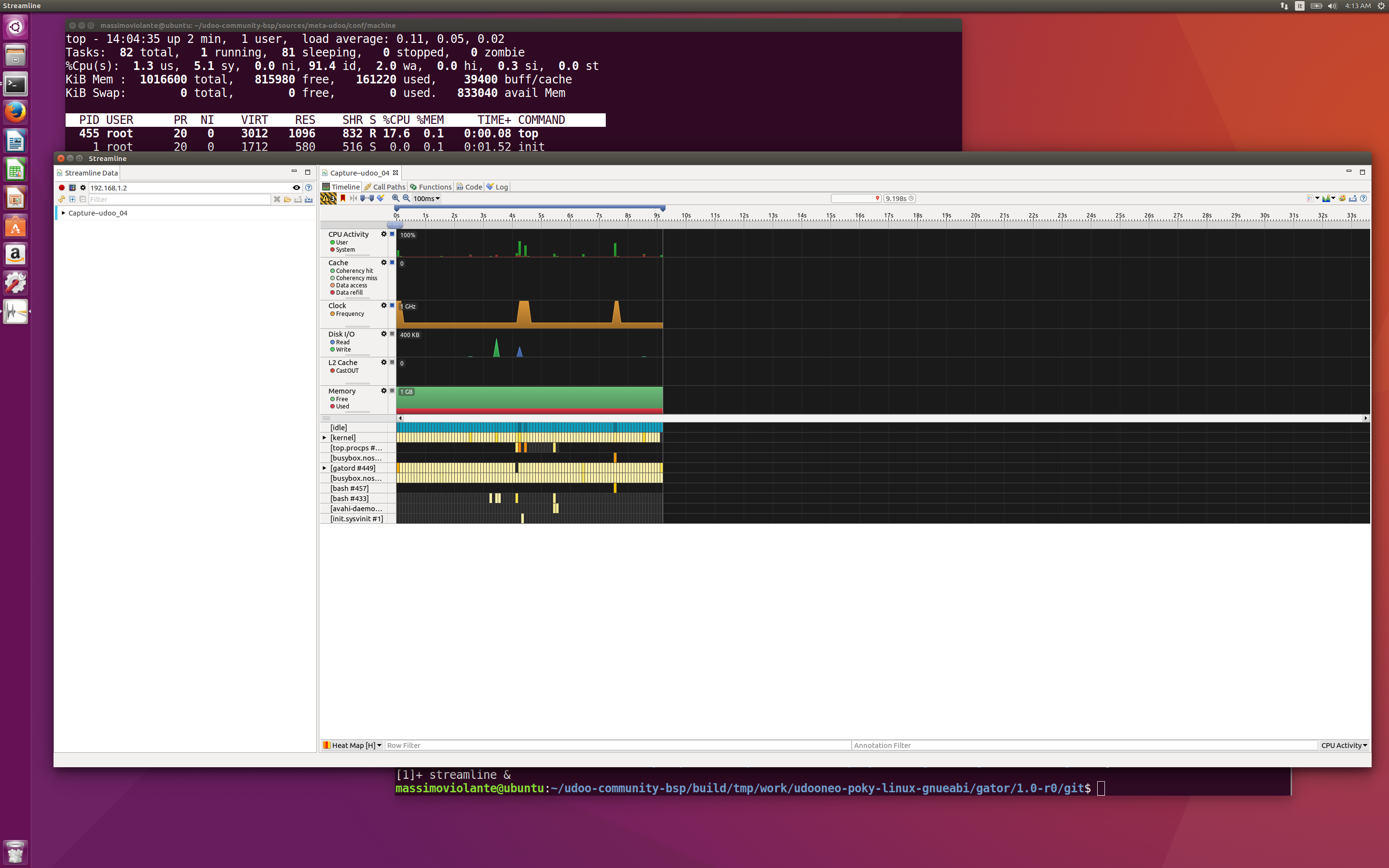
source /etc/environment

Then you should be able to call Streamline directly:

Development-host@ubuntu:~/$ streamline &

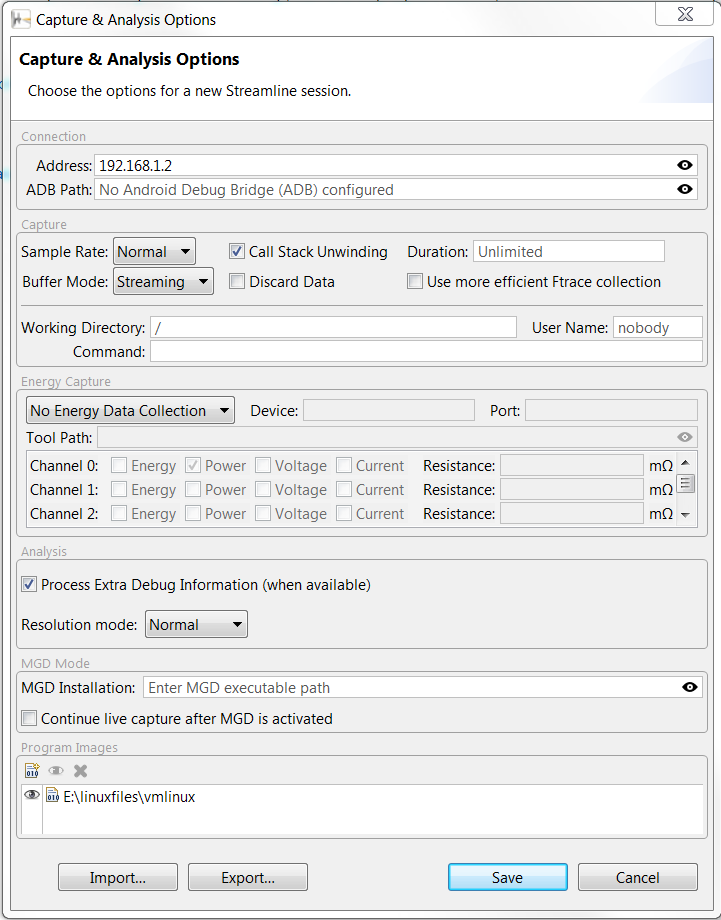
Or if you are using a Windows machine, find and run the Streamline (DS-5 CE x.xx.x)

Then, click on the **Capture & analysis button and configure** button:



In the dialogue window, configure the address for the connection, setting to 192.168.1.2 as shown below, then click on the **Add elf image** button on the Program images section (first icon on the left), and add the elf image of the Linux kernel located at:

***/home/user/raspberryPi3/rpi-build/tmp/work/raspberrypi3-poky-linux-gnueabi/linux-raspberrypi/1\_4.1.21+gitAUTOINC+ff45bc0e89-r0/linux-raspberrypi3-standard-build/vmlinux***



Streamline uses the symbols in the elf files to decode the profile timeline collected on the target, to display the collected data in a useful format.

Once done, click on **Save**.

You can now click on **Start capture**, and after specifying where to save the collected timeline, you can see that the main window of Streamline starts displaying Raspberry Pi 3 activity.

When Streamline is running, on the target console, run the ***top*** command and obtain the following output:

top - 14:04:35 up 2 min, 1 user, load average: 0.11, 0.05, 0.02

Tasks: 82 total, 1 running, 81 sleeping, 0 stopped, 0 zombie

%Cpu(s): 1.3 us, 5.1 sy, 0.0 ni, 91.4 id, 2.0 wa, 0.0 hi, 0.3 si, 0.0 st

KiB Mem : 1016600 total, 815980 free, 161220 used, 39400 buff/cache

KiB Swap: 0 total, 0 free, 0 used. 833040 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

455 root 20 0 3012 1096 832 R 17.6 0.1 0:00.08 top

1 root 20 0 1712 580 516 S 0.0 0.1 0:01.52 init

2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd

3 root 20 0 0 0 0 S 0.0 0.0 0:00.03 ksoftirqd/0

4 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kworker/0:0

5 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kworker/0:0H

6 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kworker/u2:0

7 root 20 0 0 0 0 S 0.0 0.0 0:00.04 rcu\_preempt

8 root 20 0 0 0 0 S 0.0 0.0 0:00.00 rcu\_sched

9 root 20 0 0 0 0 S 0.0 0.0 0:00.00 rcu\_bh

10 root rt 0 0 0 0 S 0.0 0.0 0:00.00 migration/0

11 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 khelper

12 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kdevtmpfs

13 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 writeback

14 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 bioset

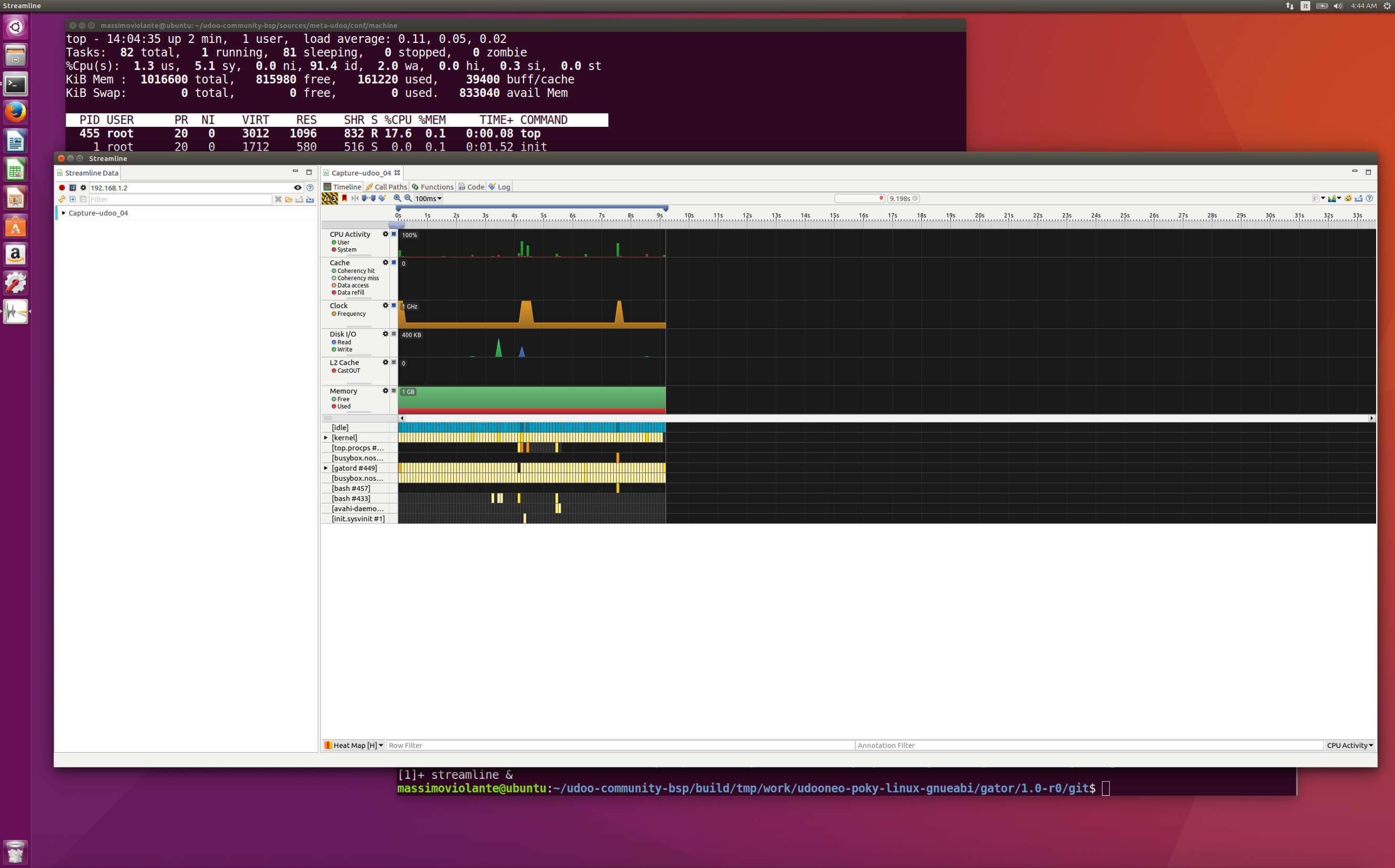
15 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 crypto

16 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kblockd

root@raspberrypi3:~#

While ***top*** is running on the Raspberry Pi 3, you can recognize increased system activity on the Streamline timeline (in the timeline below, top was executed from second 4 to about second 5).

After collecting few seconds of data, click on **Stop**. The obtained timeline shall look like as follows (please refer to the Streamline manual for details on the obtained output):



# Post-lab practice

In the previous lab, you have successfully enabled and used the debugger for a given program; the GNU debugger provides a wide range of functionalities, and we will be using them extensively in this session. In addition to the GNU debugger, you have also used streamline for resource monitoring in the current lab. This is particularly useful when examining execution time, memory usage, the clock frequency for a running program.

Now run the executable ***hello*** (or ***helloChallenge***) file that is provided with this challenge; this program is compiled with debugging features enabled. Note that the program is completely different from the one used in the Lab. Try to answer questions with the help of GNU debugger and streamline.

**1. Use appropriate tool to identify the program, explain its functionality and operation.**

A: The program is given as an executable; however, you can always view the source code using the “list” command from GNU debugger. Assuming your image is from the previous lab with debugging features enabled, simply run it on your Raspberry Pi 3:

root@raspberrypi3:~# gdbserver localhost:2000 hello

Then, set up your host by typing in terminal:

. /opt/poky/2.1.3/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi

And start the remote debugger:

~/hello-dbg$ arm-poky-linux-gnueabi-gdb hello

GNU gdb (GDB) 7.9.1

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 "--host=x86\_64-pokysdk-linux --target=arm-poky-linux-gnueabi".

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 hello...done.

**(gdb) target remote 192.168.1.2:2000**

Remote debugging using 192.168.1.2:2000

warning: Unable to find dynamic linker breakpoint function.

GDB will be unable to debug shared library initializers

and track explicitly loaded dynamic code.

0x76fcfac0 in ?? ()

(gdb)

Now you can view the source code by typing:

(gdb) list

2

3 unsigned long long operation(unsigned int x)

4 {

5 if(x == 0)

6 return 0;

7 else if (x == 1)

8 return 1;

9 else

10 return (operation(x-1)+operation(x-2));

11 }

12

13 unsigned long long main()

14 {

15 unsigned long long num;

16 printf("Enter the input number: ");

17 scanf("%llu", &num);

18 printf("%llu\n", operation(num));

19 return 0;

20 }

Upon viewing the code, the main function only asks for an input and passes it to the “operation” function. The “operation” function is non-tail-recursive with two base cases when x=0 and x=1; according to the mathematical definition, the program will calculate the Fibonacci series for the given input number.

**2. Try the program with the input equals 30, 35, 40, and 45, respectively. Record the time taken for each execution; any findings from it?**

To conveniently record and monitor the time taken for executing a program, we shall use Streamline from the main lab body. Assuming you have completed the main lab and your Linux image has the “gator.ko” kernel module included, type the following commands to set it up:

root@raspberrypi3:/# insmod /etc/init.d/gator.ko

root@raspberrypi3:/# /etc/init.d/rungator.sh

Now you are ready to start Streamline and begin monitoring the operation. Open Streamline and follow the same steps as in the main lab to begin a new capture session. Then, execute the ***hello*** program with required input numbers; you can count the execution time by monitoring CPU activity and the Clock frequency. Here are example measurements and a corresponding plot (note that this may vary slightly):

Upon inspecting the plot above, it is clear that the execution time increases exponentially with the input number. The curve becomes significantly steeper as input number increases and the trend becomes observable from around 35.

**3. You should have observed increase in time taken as the input number increases. Why would this happen? How many factors are accounted for the increase in execution time?**

There are two main factors that caused the exponential growth: input number and function call overhead. Since the program is implemented with non-tail-recursion, the operation function is called each time when calculating an element in the Fibonacci sequence. Each function call requires additional stack space and time to store the previous function and variables in it. This becomes more significant and obvious as the recursion gets deeper, and hence becomes unmanageable. Under extreme circumstances, a stack overflow may occur.

**4. Justify your answer to the previous question using GNU debugger.**

To justify the previous answer, we shall visualize and see the function calls. Firstly, we shall run the program with remote debugging:

~/hello-dbg$ arm-poky-linux-gnueabi-gdb hello

GNU gdb (GDB) 7.9.1

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 "--host=x86\_64-pokysdk-linux --target=arm-poky-linux-gnueabi".

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 hello...done.

**(gdb) target remote 192.168.1.2:2000**

Remote debugging using 192.168.1.2:2000

warning: Unable to find dynamic linker breakpoint function.

GDB will be unable to debug shared library initializers

and track explicitly loaded dynamic code.

0x76fcfac0 in ?? ()

(gdb)

You can view the source code by typing:

(gdb) list

2

3 unsigned long long operation(unsigned int x)

4 {

5 if(x == 0)

6 return 0;

7 else if (x == 1)

8 return 1;

9 else

10 return (operation(x-1)+operation(x-2));

11 }

12

13 unsigned long long main()

14 {

15 unsigned long long num;

16 printf("Enter the input number: ");

17 scanf("%llu", &num);

18 printf("%llu\n", operation(num));

19 return 0;

20 }

Now you can insert a break point at line 10, at which the program will stop temporarily:

(gdb) break 10

Breakpoint 1 at 0x104f4: file hello.c, line 10.

With the help of the break point, you can continue the program by typing continue or c; meanwhile, you can also use backtrace or bt to show the function calls. Constantly typing these two commands will allow you to examine the number of function calls and how it changes with the input number. You can try, for example, input number 3, input number 5, and input number 7, and the number of functions calls will not increase linearly but exponentially.

Evidently and intuitively, for when the input number equals, for example, 3, operation(3) is called once, operation(2) is called once, operation(1) is called twice; when the input number equals 4, operation(4) is called once, operation(3) is called once, operation(2) is called twice, and operation(1) is called 4 times.

On the other hand, if you are familiar with big O notation, by inspecting the last line of code, it will become clear that:

O(2n-1) + O(2n-2) + O(1) = O(2n)

**5. How much memory does one function call occupy? Use your GNU debugger to find out a specific value.**

Moving on from the previous question, a very useful command from GNU debugger is ***info frame***:

(gdb) info frame 1

Stack frame at 0x7efffca8:

pc = 0x10508 in operation (hello.c:10); saved pc = 0x10508

called by frame at 0x7efffcc0, caller of frame at 0x7efffc90

source language c.

Arglist at 0x7efffc90, args: x=<optimized out>

Locals at 0x7efffc90, Previous frame's sp is 0x7efffca8

Saved registers:

r4 at 0x7efffc90, r5 at 0x7efffc94, r6 at 0x7efffc98, r7 at 0x7efffc9c, r8 at 0x7efffca0, lr at 0x7efffca4

(gdb) info frame 2

Stack frame at 0x7efffcc0:

pc = 0x10508 in operation (hello.c:10); saved pc = 0x10374

called by frame at 0x7efffcd0, caller of frame at 0x7efffca8

source language c.

Arglist at 0x7efffca8, args: x=<optimized out>

Locals at 0x7efffca8, Previous frame's sp is 0x7efffcc0

Saved registers:

r4 at 0x7efffca8, r5 at 0x7efffcac, r6 at 0x7efffcb0, r7 at 0x7efffcb4, r8 at 0x7efffcb8, lr at 0x7efffcbc

By subtracting the addresses of the two adjacent stack frames, it turns out that one stack frame occupies 24 addresses, and hence 24 bytes. **Check: “info stack” at: https://stackoverflow.com/questions/37321252/check-used-stack-size-using-core-file**

**6. Suggest and implement a new approach to solve the above timing issue (Hint: think about the recursion and iteration, you can reuse the Makefile from the Lab session).**

Since the majority of the increase in execution time is due to function call overhead, we can eliminate it by replacing the recursion method with iteration method. Here is an example solution:

#include <stdio.h>

unsigned long long operation(unsigned int n)

{

unsigned long long a = 0;

unsigned long long b = 1;

unsigned long long temp;

while (n != 0)

{

temp = a;

a = b;

b = b + temp;

n--;

}

return a;

}

unsigned long long main()

{

unsigned int num;

printf("Enter the input number: ");

scanf("%llu", &num);

printf("%llu\n", operation(num));

return 0;

}

**7. Repeat step 2 and comment on the improvement of your new program.**

Now make your program as you did in the previous lab with debug features included, run it with/without remote debugging and use your Streamline to record the execution time.

The iteration method has the time complexity of O(n) and memory complexity of O(1). In addition, since only one function is called during the computation process, the program does not suffer from significant function call overhead. Therefore, the program responses quickly for those given inputs.

**8. Now try your new program with 91, 92, 93, and 94 as the input number; any error inspected? What might be the cause of the potential error?**

Assume you have defined variables in unsigned long long, you should get the following output:

|  |  |
| --- | --- |
| **Input Number** | **Output** |
| 91 | 4,660,046,610,375,530,309 |
| 92 | 7,540,113,804,746,346,429 |
| 93 | 12,200,160,415,121,876,738 |
| 94 | 1,293,530,146,158,671,551 |

From the table above, it is clear that the output number for input 94 is incorrect. This is called overflow—the original output number is greater than the maximum of an unsigned long long can contain, which is 64-bit.

Theoretically, the maximum for an unsigned long long is 2^64-1= 18,446,744,073,709,551,615; hence, when the output for the 94th Fibonacci number is greater than this, the most significant bit in binary is missed, leaving the rest bits that are incorrect. Therefore, the overflow is the cause of the error.

**9. With the help of the GNU debugger, justify the cause of the above error.**

Assuming you have correctly analyzed the cause of above error as overflow, we shall visualize it with GNU debugger. The overflow occurs when the last addition operation is conducted and Arm is a Load-Store architecture; hence, the overflow will only occur in registers at the moment when the last addition operation is conducted. Therefore, by showing the content of registers at an appropriate time, we shall inspect the overflow in registers that caused the error.

First, run your program with GNU debugger as follows:

~/hello-dbg$ arm-poky-linux-gnueabi-gdb hello

GNU gdb (GDB) 7.9.1

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 "--host=x86\_64-pokysdk-linux --target=arm-poky-linux-gnueabi".

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 hello...done.

**(gdb) target remote 192.168.1.2:2000**

Remote debugging using 192.168.1.2:2000

warning: Unable to find dynamic linker breakpoint function.

GDB will be unable to debug shared library initializers

and track explicitly loaded dynamic code.

0x76fcfac0 in ?? ()

(gdb)

Then, use the following command to show the disassembly view of the program:

(gdb) disas /m main

Dump of assembler code for function main:

warning: Source file is more recent than executable.

6 unsigned long long a = 0;

0x000103d0 <+136>: mov r4, #0

0x000103d4 <+140>: mov r5, #0

0x000103d8 <+144>: b 0x103ac <main+100>

7 unsigned long long b = 1;

8 unsigned long long temp;

9

10 while (n != 0)

0x00010370 <+40>: cmp r3, #0

0x00010374 <+44>: beq 0x103d0 <main+136>

0x00010378 <+48>: mov r4, #1

0x0001037c <+52>: mov r5, #0

0x00010380 <+56>: mov r0, #0

0x00010384 <+60>: mov r1, #0

0x00010388 <+64>: b 0x10394 <main+76>

0x000103a0 <+88>: subs r3, r3, #1

0x000103a4 <+92>: mov r1, r5

0x000103a8 <+96>: bne 0x1038c <main+68>

11 {

12 temp = a;

13 a = b;

14 b = b + temp;

0x0001038c <+68>: mov r4, r6

0x00010390 <+72>: mov r5, r7

0x00010394 <+76>: adds r6, r0, r4

0x00010398 <+80>: mov r0, r4

0x0001039c <+84>: adc r7, r1, r5

15 n--;

16 }

17

18 return a;

19 }

20

21 unsigned long long main()

22 {

0x0001034c <+4>: push {r4, r5, r6, r7, lr}

0x00010354 <+12>: sub sp, sp, #12

23 unsigned int num;

24 printf("Enter the input number: ");

0x00010348 <+0>: movw r0, #1476 ; 0x5c4

0x00010350 <+8>: movt r0, #1

0x00010358 <+16>: bl 0x1030c <printf@plt>

25 scanf("%du", &num);

0x0001035c <+20>: movw r0, #1504 ; 0x5e0

0x00010360 <+24>: movt r0, #1

0x00010364 <+28>: add r1, sp, #4

0x00010368 <+32>: bl 0x10330 <\_\_isoc99\_scanf@plt>

26 printf("%llu\n", operation(num));

0x0001036c <+36>: ldr r3, [sp, #4]

0x000103ac <+100>: movw r0, #1512 ; 0x5e8

0x000103b0 <+104>: mov r2, r4

---Type <return> to continue, or q <return> to quit---

0x000103b4 <+108>: mov r3, r5

0x000103b8 <+112>: movt r0, #1

0x000103bc <+116>: bl 0x1030c <printf@plt>

27 return 0;

28 } 0x000103c0 <+120>: mov r0, #0

0x000103c4 <+124>: mov r1, #0

0x000103c8 <+128>: add sp, sp, #12

0x000103cc <+132>: pop {r4, r5, r6, r7, pc}

End of assembler dump.

From the assembly code, it is clear that the variable ***a***, which is returned as the result of the computation, is represented by registers r4 and r5 (since long long contains 64 bits and Arm is 32-bit architecture). The r4 and r5 registers are initialized as 0.

Then, you can insert a break point at instruction 0x0001039c by using:

break \*0x0001039c

With the break point, your program will stop whenever the break point is reached. Since the instruction is in a while loop and we are intended to observe the register overflow, we shall set the break point as “conditional” by ignoring the first few times until the overflow is about to happen:

ignore 1 45

The above command will tell the debugger to ignore break point 1 for the first 45 times; therefore, the program will only stop at the 46th time encountering the break point.

Now run the program with the command:

continue

or simply type ***c***. Then, the Raspberry Pi 3 will ask you for an input number, it is good to input either 96 or larger.

When the program stops at the break point, we shall visualize the content in registers by:

Breakpoint 1, 0x0001039c in operation (n=51) at operation.c:14

14 b = b + temp;

(gdb) info registers

r0 0x6d73e55f 1836311903

r1 0x0 0

r2 0x1 1

r3 0x33 51

r4 0x6d73e55f 1836311903

r5 0x0 0

r6 0xb11924e1 2971215073

r7 0x0 0

r8 0x0 0

r9 0x0 0

r10 0x76fff000 1996484608

r11 0x0 0

r12 0x1 1

sp 0x7efffcb0 0x7efffcb0

lr 0x1036c 66412

pc 0x1039c 0x1039c <main+84>

cpsr 0x900d0010 -1878196208

From the value stored in r4, it is clear that the current result is 1836311903, which is indeed the 46th Fibonacci number. On the other hand, the value of r5 is 0.

Now go on executing the program by issuing the following commands:

continue

info registers

The output from console will be:

Breakpoint 1, 0x0001039c in operation (n=50) at operation.c:14

14 b = b + temp;

(gdb) i r

r0 0xb11924e1 2971215073

r1 0x0 0

r2 0x1 1

r3 0x32 50

r4 0xb11924e1 2971215073

r5 0x0 0

r6 0x1e8d0a40 512559680

r7 0x0 0

r8 0x0 0

r9 0x0 0

r10 0x76fff000 1996484608

r11 0x0 0

r12 0x1 1

sp 0x7efffcb0 0x7efffcb0

lr 0x1036c 66412

pc 0x1039c 0x1039c <main+84>

cpsr 0x200d0010 537722896

Note that the value in r4 is 2971215073, which is indeed the 47th Fibonacci number. Then, continue the program:

continue

info registers

The output now is:

Breakpoint 1, 0x0001039c in operation (n=49) at operation.c:14

14 b = b + temp;

(gdb) i r

r0 0x1e8d0a40 512559680

r1 0x0 0

r2 0x1 1

r3 0x31 49

r4 0x1e8d0a40 512559680

r5 0x1 1

r6 0xcfa62f21 3483774753

r7 0x1 1

r8 0x0 0

r9 0x0 0

r10 0x76fff000 1996484608

r11 0x0 0

r12 0x1 1

sp 0x7efffcb0 0x7efffcb0

lr 0x1036c 66412

pc 0x1039c 0x1039c <main+84>

cpsr 0x800d0010 -2146631664

If we look at the value in r4, the value stored is 512559680, which is inconsistent with the 48th Fibonacci number and smaller than it should be. The reason is that each register is 32-bit long, and the 48th Fibonacci number is 4807526976, exceeding the range covered by an unsigned 32-bit number. However, since we declared the value a as unsigned long long, the variable is stored in two registers with 64 bits available in total. Therefore, the carry is stored in r5, with most significant bits in r5 and least significant bits in r4.

We can prove it by combining these two registers: 0x00000001 and 0x1e8d0a40. The result is a 64-bit number: 0x000000011e8d0a40, which is equivalent to decimal: 4807526976, indeed the 48th Fibonacci number.

Following the same philosophy, we ignore the next 45 iterations until the next overflow in r5 occurs:

ignore 1 45

continue

info registers

The corresponding output is:

Breakpoint 1, 0x0001039c in operation (n=4) at hello.c:14

14 b = b + temp;

(gdb) i r

r0 0x221f2702 572466946

r1 0x68a3dd8e 1755569550

r2 0x1 1

r3 0x4 4

r4 0x221f2702 572466946

r5 0xa94fad42 2840571202

r6 0x840bf6bf 2215376575

r7 0xa94fad42 2840571202

r8 0x0 0

r9 0x0 0

r10 0x76fff000 1996484608

r11 0x0 0

r12 0x1 1

sp 0x7efffcb0 0x7efffcb0

lr 0x1036c 66412

pc 0x1039c 0x1039c <main+84>

cpsr 0x900d0010 -1878196208

Now the values are: r4=ox221f2702 and r5=0xa94fad42. If we combine them as before, the decimal equivalent is: 12200160415121876738, which is the 93rd Fibonacci number.

Now as we know, the 94th Fibonacci number is too large to be stored in a pair of registers, the next ought to cause overflow leading to an incorrect result:

continue

info registers

And the output is:

Breakpoint 1, 0x0001039c in operation (n=3) at hello.c:14

14 b = b + temp;

(gdb) i r

r0 0x840bf6bf 2215376575

r1 0xa94fad42 2840571202

r2 0x1 1

r3 0x3 3

r4 0x840bf6bf 2215376575

r5 0x 301173456

r6 0xa62b1dc1 2787843521

r7 0x11f38ad0 301173456

r8 0x0 0

r9 0x0 0

r10 0x76fff000 1996484608

r11 0x0 0

r12 0x1 1

sp 0x7efffcb0 0x7efffcb0

lr 0x1036c 66412

pc 0x1039c 0x1039c <main+84>

cpsr 0x800d0010 -2146631664

Look into r4 and r5; we can combine the values, and the decimal equivalent is: 1293530146158671551, an incorrect result due to register overflow and identical to what we get in the previous question.

**10. Fix the above error and test it again.**

#include <stdio.h>

#include <stdlib.h>

/\*Count the number of bits\*/

unsigned int bit\_check(unsigned long long n)

{

unsigned int bits =0;

while (n != 0)

{

bits++;

n = n >> 1;

}

return bits;

}

/\*Check if the addition is safe\*/

int overflow\_check (unsigned long long a ,unsigned long long b)

{

if (bit\_check(a) < 64 && bit\_check(b) <= 64)

return 1;

else

return 0;

}

unsigned long long operation(unsigned int n)

{

unsigned long long a = 0;

unsigned long long b = 1;

unsigned long long temp;

while (n != 0)

{

temp = a;

a = b;

if (overflow\_check(temp,b))

{

b = b + temp;

}

else

{

printf("Potential overflow detected: Computation terminating\n ");

exit(1);

}

n--;

}

return a;

}

unsigned long long main()

{

unsigned int num;

printf("Enter the input number: ");

scanf("%du", &num);

printf("%llu\n", operation(num));

return 0;

}

In the above example solution, we have implemented two functions to count the number of bit a given and check potential overflow. Since the variable b is “one step ahead” of the result in the operation, it is then safe having 64 bits.