# Assignment 4 A Barrier Mechanism in Linux Kernel

team 20: Kausic (1213203730), beibei (1210467866) This document goes into detail about the process of implementing the mechanism of barrier synchronizations.

## Contents:

The repository contains 4 files: 1) a patch file: consists of changes to the original kernel source code 2) a testing program main.c 3) a Makefile (to make the testing program) 4) a README file

## Compile Instructions:

Once you download the files, navigate into the directory that contains the 4 files. Now it is assumed that the sdk is already installed in your computer. The default directory for the set up is /opt/iot-devkit/1.7.2/sysroots/ which will contain the kernel and the i586-poky-linux-gcc. If you have installed your sdk in some other directory then make sure the path variable leads to your i586-poky-linux-gcc compiler and the make is given the right directory to the kernel.

**First, apply the patch file:**

Change the directory to the directory of the intact kernel downloaded, type: patch -p1 < <assignment4.patch> . <assignment4.patch> is the directory of patch file. Then the kernel is patched.

**Then, compile the kernel:**

**1) go to the kernel directory:**

Type*: vim bashrc* , go to the bottom, add: export PATH=/opt/iot-devkit/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux:$PATH *source ~/.bashrc* or you can just type in the terminal every time: *export PATH=/opt/iot-devkit/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux:$PATH*

**2) put the following line in Makefile or just type:**

*ARCH=x86 LOCALVERSION= CROSS_COMPILE=i586-poky-linux- make -j4* It takes some time to compile the kernel.

**3) put the following line in Makefile or just type:**

*ARCH=x86 LOCALVERSION= INSTALL_MOD_PATH=../galileo-install CROSS_COMPILE=i586-poky-linux- make modules_install* Then the bzImage should be ready in the directory "kernel/arch/x86/boot/bzImage".

**4) copy the ready bzImage from the directory "kernel/arch/x86/boot/bzImage", put it in to SD card, replacing the original bzImage in SD card, reboot it.**

**For the user space program:**

go to the directory of where the user c program is saved, type: *make* then a "Test" file is generated in the same directory. Once you get this file, You will have to transfer the file to the galileo board. type: *sudo scp Test root@192.168.1.5:/home/* Replace the ip address with the static IP you have assigned to your board. To run the user program, from the directory where you have the Test binary in the galileo, type: *./Test > log.txt* You may be asked to input

the average sleep time. The recommeded value is below 15. The macro TIMEOUT can be changed to set the timeout. We recommend a value greater than 3000 After input, please put "enter". Then you will be able to see the outcome of the execution.

## Verifying our barrier method:

Now, if you run the Test executable without redirecting it to the log file It will print out a lot of information which we might not be able to make sense of. Hence, we print it to a log file.

**./Test > log.txt**

Our user program is in three parts:

- The first part initiates two barriers. The first barrier is excercised using 5 threads and the second barrier is using 20 threads and both barrier are excercised for 100 iterations. Now, we have specified a timeout value which does not produce timeout and works well.
- the second part demonstrates the barrier destroy mechanism. We destroy the first barrier we initiated in the previous step and use it to run 5 threads for 10 iterations each.
- The third part of the program demonstrates the timeout functionality. It creates a seperate barrier and intialized a small timeout value for the barrier and executes 20 threads with 10 iterations each.

The kernel messages will be printed on the stdout. You can see that once the initialization system call is called . This will show the PID of the calling process and the barrier ID initialized as well as the address of the barrier structure.

In the log file, after the statement Testing our barrier it will show the output of the thread that prints the round number, thread ID, Process ID and Barrier ID. You can see that the threads are organised.As in for a given process number, all the threads would have completed thier round and only then will it proceed to the next round. This shows that our barrier wait mechanism works correctly.

For the second part,in the log file if you see after " Now demonstrating barrier destroy" You can see the same output being printed, but the threads won't be organised. This shows that the threads won't wait for the other threads in any manner thus proving our barrier destroy works.

For the third part, in the log file if you see after " Now demonstrating timeout functionality" you can see the output being printed but there will be occasional timoouts being printed . Once a timeout happens the barrier is reset for the next iteration. This proves our timeout functionality also works. When the timeout occurs a handler will also print the kernel message that handler has been called.