Assignment 3 A Barrier Mechanism in Linux Kernel (150 points)

Exercise Objectives

1. To learn the basic programming technique in Linux kernel.
2. To learn the software structures of kernel thread and synchronization.
3. To practice the mechanisms and data structures of system call, wait queue, and other Linux kernel objects

Project Assignment

Barrier synchronization has been applied widely in parallel computing to synchronize the execution of parallel loops. The basic idea is to allow parallel loop computation to start next iteration if all loops have completed the previous iteration. In NPTL pthread library, it is realized by 3 functions for threads of the same process, i.e., pthread_barrier_init, pthread_barrier_wait, and pthread_barrier_destroy. A description of the functions can be found in https://docs.oracle.com/cd/E19253-01/816-5137/gfwek/index.html, and their source code is available in glibc NPTL cross reference page http://code.metager.de/source/xref/gnu/glibc/nptl/.

In this assignment, you are required to develop a barrier synchronization mechanism in Linux kernel which can be invoked by user processes via system call interface. The mechanism consists of 3 Linux kernel system calls which are resemble to the pthread barrier:

1. *barrier_init(unsigned int count, unsigned int *barrier_id)* – to initiate a barrier in the caller's address space. "count" is the number of threads (i.e., processes and lightweight processes) that are required to be synchronized. If the barrier is initiated successfully, an integer barrier id is returned to the caller process.
2. *barrier_wait(unsigned int barrier_id)* – to wait for a barrier synchronization. The caller process will be blocked until the required number of threads have called the wait function.
3. *barrier_destroy(unsigned int barrier_id)* – to destroyed the barrier of the id *barrier_id.*

Your implementation should meet the following requirements:

- The calls should return proper error status if the required operations cannot be completed.
- Each barrier should be initiated in caller's address space and can only be used by the processes sharing the space.
- *barrier_id* is unique in its address space and the *barrier_id*'s of different address space may have the same value.
- Once a barrier is initiated, it can be used for consecutive synchronization operations until it is destroyed.
- You can choose to implement a barrier which either blocks or allows a *barrier_wait* call to enter a new round of synchronization while some of the waiting processes of the previous round are waked up.
- The barrier mechanism must work properly in SMP systems and allow multiple user processes work on multiple barriers in their corresponding address space concurrently. Unfortunately, Quark doesn't have multiple cores. Thus, we cannot test this execution environment.

To demonstrate your barrier implementation, a testing program should be developed. The testing program forks two child processes. In each child process, there are 5 threads to exercise the 1st barrier and additional 20 threads use the 2nd barrier. Each thread sleeps a random amount of time before entering a new round of synchronization. The average sleep time in microsecond should be an input integer to the testing program. The testing program terminates after 100 rounds of synchronization on

each barrier in each child process. To wait for the completion of child processes, your testing program can use "wait" system call. An example can be found in http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/wait.html.

Since Linux system calls are statically defined and compiled into the kernel, you will need to rebuilt the kernel with new system calls. You may consider the following suggested procedure to rebuild 3.8 Linux kernel for Galileo Gen2 board:

- Use the pre-built toolchain and the Linux source linux3-8-r0.tar.bz2 from Dropbox/CSE438-530-Gen2. Also, rename the file dot_config from the folder as ".config" and use it for kernel building.
- Include the cross-compilation tools in your PATH:

  export PATH=path_to_sdk/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux:$PATH
- Cross-compile the kernel

  ARCH=x86 LOCALVERSION= CROSS_COMPILE=i586-poky-linux- make -j4
- Build and extract the kernel modules from the build to a target directory (e.g ../galileo-install)

  ARCH=x86 LOCALVERSION= INSTALL_MOD_PATH=../galileo-install CROSS_COMPILE=i586-poky-linux- make modules_install
- Extract the kernel image (bzImage) from the build to a target directory (e.g ../galileo-install)

  cp arch/x86/boot/bzImage ../galileo-install/
- Install the new kernel and modules from the target directory (e.g ../galileo-install) to your micro SD card
  - Replace the bzImage found in the first partition (ESP) of your micro SD card with the one from your target directory (backup the bzImage on the micro SD card e.g. rename it to bzImage.old)
  - Copy the kernel modules from the target directory to the /lib/modules/ directory found in the second partition of your micro SD card.
- Reboot into your new kernel

Note that, after the first build, there is no need to rebuild/replace the kernel modules if you don't make any changes in the loadable modules. Also, to avoid building the kernel for each code revision, you may implement the functions in a loadable driver module and use device file operations to invoke them. Once the functions are fully developed, you can then add them as system calls and rebuild the kernel.

Due Date

The assignment is due at 11:59pm, Nov. 13.

What to Turn in for Grading

- The submission includes a patch file that consists of changes to the original kernel source code, a testing program, a Makefile (to make the testing program), and a README file.
- There will be 5 bonus points each day if you submit earlier (a maximum of 20 bonus points for the part) and 20 points penalty per day if the submission is late.
- Failure to follow these instructions may cause an annoyed and cranky TA or instructor to deduct points while grading your assignment.
- Here are few general rule for deductions:
  - No make file or compilation error -- 0 point for the part of the assignment.
  - Must have "–Wall" flag for compilation -- 5-point deduction for each warning.
  - 10-point deduction if no compilation or execution instruction in README file.
  - Source programs are not commented properly -- 10-20-point deduction.