Real Time System
Project#1

Team7 - R04922058 , R04922034

1 Part I: Linux Kernel Building

Kernel Building Steps

In this section, the fundamental steps required to build a kernel are examined.

1. `Install the Requried Pacakges && Get the Source Code`
   'sudo apt-get install fakeroot build-essential kernel-package libncurses5 libncurses5-dev'

   - fakeroot:
     fakeroot is a package that allows a normal user to fake itself as root in order to prevent the need of changing permissions of files when building an archive. fakeroot start up a shell and fake all the permission related such as chown , chgrp operation by simulation, and nothing happens to the actual permission of the files. When building a kernel package, we require some of the scripts and codes to be owned root, but we don't want to be limited to archive the package with root permission, and fakeroot comes into use.
   - build-essential:
     build-essential is a set of tools that is required to build a debian package, which are defined in the debian policy manual. It is related to a set of compilers as well as GNU Libraries. A few examples are : dpkg-dev ,g++ , gcc ,libc and make.
   - kernel-package:
     This package is capable of handling the steps required to compile a kernel using make-kpkg, and it also helps to maintain multiple version of kernels in a single machine without fear of contaminating one another. It also allows user to create kernel-headers.
   - libncurses5 and libncurses5-dev:
     libncurses libraries are used to render graphical user interface on terminal, such as the one we see when executing 'make menuconfig'.

2. `Clean up the Source Code`
   'make mrproper'
   Clean up all the settings as well as the temporary files such that we get a clean source code. 'make mrproper' will remove the .config generated by 'make menuconfig' as well as the tmporary .o files. This is different from 'make clean', which removes the .o files only.

3. `make menuconfig`
   make menuconfig will read the Kconfig files,presents a libncureses generated user interface for user to configure the kernel and save the final configuration in .config file.
   Possible configurations for each module are

- `[*] include`
  Included modules will be compiled into the kernel.
- `[ ] exclude`
  Excluded modules will not be compiled into the kernel , nor will them be compiled separately into loadable modules.
- `[M] modularized`
  Modularized modules will be compiled in the 'make module' step and installed in the 'make module install' step.

4. `Compile the Kernel`
   'make bzImage'
   make bzImage will create a bzImage file under /arch/$ARCH/ where bzImage stands for big zImage that is compressed by gzip, not bzip. This is loaded by the boot-loader onboot.

5. `Compile the Modules`
   'make modules' This step compiles all modules that are configured as modularized in the 'make menuconfig' step.

6. `Install the Modules to Kernel`
   'make module_install' This step copies all the compiled modules into the lib/modules/$kernel_version/ directory.

7. `Setup the Kernel for Boot`
   'make install'
   make install will create System.map, initrd.img and config under the directory /boot. The bzImage file generated at 'make bzImage' step will be copied to /boot/ and renamed as vmlinuz. The older version of the kernel will be renamed as vmlinuz.old

   - vmlinuz is a compressed version of vmlinux by zlib, which is decompressed at boot up. [1]
   - initrd.img is a minimum Linux ram-disk image that is mounted by the bootloader to mount all other modules and execute /sbin/init.
   - System.map is a symbol table used by the kernel, required by processes such as klogd , lsof , ps and so on. config is a copy of the .config file.

8. `Update Grub for future Booting`
   uncomment grub_hidden_timeout and update-grub2
   Uncommenting the grub_hidden_timeout related settings allow users to switch between various versions of Linux to boot into.

[1] zlib is a software library use for data compression, an abstraction of the DEFLATE compression algorithm used in gzip file compression program, there fore a simple renaming of bzImage works.

2 Part II: Linux Scheduling Policy Testing

Linux Scheduling Concepts

Linux determines which thread to run next based on each thread's `sched_priority` value. Threads scheduled using real-time policies (`SCHED_FIFO`, `SCHED_RR`) can have a priority value from 1(low) to 99(high). Priority values of normal scheduling policies (`SCHED_OTHER`, `SCHED_BATCH`, `SCHED_IDLE`) are always 0(lowest). Since the scheduling in Linux is preemptive, real-time jobs can always preempt non-real-time jobs.

By default, threads are scheduled by `SCHED_OTHER` policy, however, Linux provides some system calls for users to change the policy for a certain thread. In this project, we are asked to create threads scheduled by FIFO. Since a `SCHED_FIFO` thread can run as long as it wishes without time slicing, which may block other threads forever, it requires root privileges to activate this policy.

Implementation

We used Pthreads library to implement the multi-thread C program.

Change the scheduling policy

First, we called `setFIFO()` function to change main thread's scheduling policy to FIFO. The code snippet is shown below:

```
void setFIFO(){
  struct sched_param param;
  param.sched_priority = sched_get_priority_min(SCHED_FIFO);
  sched_setscheduler(0, SCHED_FIFO, &param);
}
```

There are two system calls in this function:

1. `sched_get_priority_min(int policy)`
   This function returns the minimum priority value that can be set using `policy`. It usually returns 1 when `policy = SCHED_FIFO`.

2. `sched_setscheduler(pit_t pid, int policy, const struct sched_param *param)`
   This function sets scheduling policy and the associated parameters for the thread specified by `pid`. `pid=0` refers to current thread.

Inherit scheduling attribute

Second, instead of individually setting the scheduling policy for each new thread, we used `PTHREAD_INHERIT_SCHE` to specify that each thread should inherit its scheduling policy from the parent.

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr, PTHREAD_INHERIT_SCHED)
pthread_create(&thread, &attr, ...);
```

Discussion

FIFO scheduling

With the FIFO scheduling policy, a thread can run as long as it wishes unless one of these conditions occurs:

1. It is blocked (ex: I/O, mutex).
2. It is preempted by a higher priority thread.
3. It volunteers to give up control of processor (ex: sleep, yield).

Therefore, we can't implement "busy for one second" by sleep() function in this project. Calling sleep will remove the current thread from runqueue even if it is scheduled by FIFO. Our solution is to use clock() function with a while loop, which continuously checks the number of clock cycles already used until a second has elapsed.

CPU affinity

Setting CPU affinity forced a thread to run on assigned CPU(s). Each CPU has its own runqueue. Without setting affinity, threads can be put in different FIFO queue and execute simultaneously, which is not the result we want in this project. A way to solve this problem is using `pthread_setaffinity_np()` function. According to Linux manual, child process will inherit parent's CPU affinity by default, so we only need to specify main thread's affinity.

Result

```
pj1@pj1-VirtualBox:~/rts$ ./sched_test
2 starts running.
get policy 2:OTHER
        Thread 2 is running
1 starts running.
get policy 1:OTHER
        Thread 1 is running
0 starts running.
get policy 0:OTHER
        Thread 0 is running
        Thread 0 is running
        Thread 1 is running
        Thread 2 is running
        Thread 1 is running
        Thread 0 is running
        Thread 2 is running
        Thread 0 is running
        Thread 1 is running
        Thread 2 is running
        Thread 0 is running
        Thread 1 is running
        Thread 2 is running
```

Figure 1: Default

```
pj1@pj1-VirtualBox:~/rts$ sudo ./sched_test SCHED_FIFO
FIFO success.
0 starts running.
get policy 0:FIFO
        Thread 0 is running
        Thread 0 is running
        Thread 0 is running
        Thread 0 is running
        Thread 0 is running
1 starts running.
get policy 1:FIFO
        Thread 1 is running
        Thread 1 is running
        Thread 1 is running
        Thread 1 is running
        Thread 1 is running
2 starts running.
get policy 2:FIFO
        Thread 2 is running
        Thread 2 is running
        Thread 2 is running
        Thread 2 is running
        Thread 2 is running
```

Figure 2: FIFO