Assignment 1. Shared Message Queues in User and Kernel Space (200 points)

Assignment Objectives

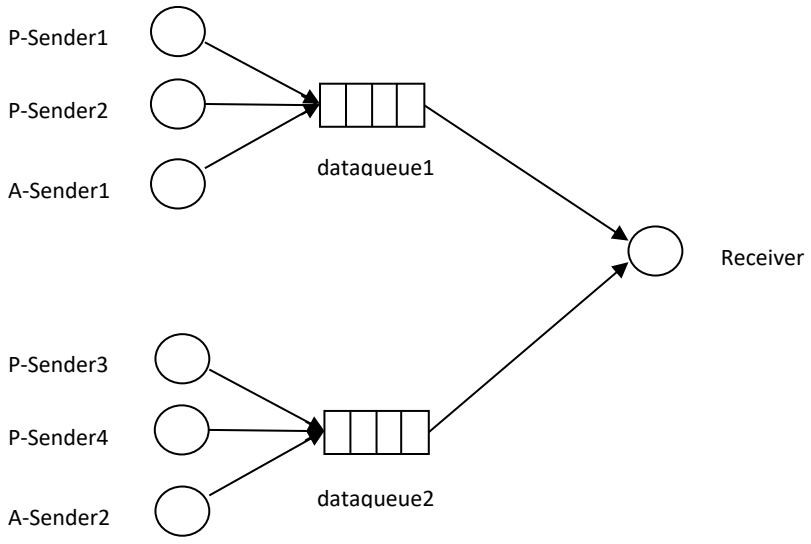
1. To learn the basic programming technique for module and device driver in Linux kernel.
2. To learn thread programming for periodic and aperiodic tasks in Linux.
3. To learn basic synchronization mechanisms in Linux user and kernel space.
4. To use x86's TSC (Time stamp counter) to measure elapse time.

Project Assignment

Message bus is a useful middleware to support the communication among user applications. The core component of a message bus is its bus daemon which manages connections and forwards messages through message queues. An example of message buses is D bus, developed as part of the freedesktop.org project for desktop applications.

In this assignment, you are required to develop few programs to run in Linux systems. The programs should include

1. A program to implement shared queues that can perform basic enqueue and dequeue operations and record the accumulated queueing time (i.e., the elapse time between enqueue and dequeue operations).
2. A test program that initiates multiple threads to access shared queues as illustrated in the following test case:



In the test program, you will create 4 periodic sender threads, 2 aperiodic sender threads, and one receiver thread, running in a user application. The threads use 2 shared FIFO queues to pass message as shown in the test case diagram. Each message is represented by a message id, source id, a queueing time, and a double-precision floating-point number. When a message is dequeued, its queueing time is computed based on the Time Stamp Counter (TSC) values of the enqueue and dequeue instants. Each queue can be shared by multiple tasks (threads or processes if it is in kernel space) concurrently and should be implemented as a ring of pointers which point to a dynamically-allocation memory region for storing message body. The default queue length is 10, and can be defined in a macro (i.e. using #define directive).

The periodic sender threads (P-sender) run periodically and their periods are multipliers of *BASE_P* where *BASE_P* is defined as:

```
// the base unit of period in micro-seconds
#define BASE_PERIOD 1000
```

The two aperiodic senders, A-sender1 and A-sender2, become active when there are mouse button press events (left and right, respectively).

Once a sender thread becomes active, it:

- Calculate π using an iterative algorithm (https://www.codeproject.com/Articles/813185/Calculating-the-Number-PI-Through-Infinite-Sequenc) and the number of iterations is a random number between 10 and 50.
- Send a message including the computed π value to the designated data queue.
- Wait for the next period or the next mouse event.

Periodically, a receiver thread performs the operations of reading all queued messages from data queues, and records the accumulated queueing time of the received message.

In your test program, shared queues should be initiated first. Then, all threads are created and start to run periodically or aperiodically with corresponding real-time priorities. The threads are scheduled with Real-time FIFO policy. The priorities of A-sender threads are higher than that of P-sender threads, which are higher than the priority of the receiver thread. The period multipliers for P-senders, and the receiver are defined in constant arrays, for instance,

```
const int p_period_multiplier[] = {12, 32, 18, 28};
const int r_period_multiplier[] = {40};
```

When a double click of mouse left button is detected, i.e., two consecutive button press events within 500ms, your program enters a termination sequence. During the sequence, all active sender and receiver threads must complete their on-going operations before exit, and all messages enqueued must be received. Finally, the main program can exit after printing out the total number of messages received at the receiver thread, the average and standard deviation of message queueing times.

Part 1 – Shared queues in a library

In the first part of the assignment, the shared queues are implemented as a library unit in user space. It must contain 4 functions and be thread-safe (can be called by multiple threads concurrently):

- *sq_create*: to create a shared queue of message with a specific length.
- *sq_write*: to enqueuer a message. -1 is returned if the queue is full.
- *sq_read*: to dequeuer a message, -1 is returned if the queue is empty.
- *sq_delete*: to delete a shared queue.

Part 2 – Shared queues as kernel devices

In this part, the shared queues are implemented in kernel space and managed by device driver “squeue”. To avoid the dynamic creation of shared queues, four queue devices, named “dataqueue1” and “dataqueue2” are created and added to Linux device file systems when the device driver for the shared queues is installed. The devices act as FIFO (first-in-first-out) queues where messages can be enqueued and dequeued. The device driver should implement the following file operations:

- *open:* to open a device (the devices are "dataqueue1", and "dataqueue2").
- *write:* to enqueue a message to the device. If the queue is full, -1 is returned and errno is set to EINVAL.
- *read:* to dequeue a message (with the valid queueing time) from the device and copy it to the user buffer. If the queue is empty, -1 is returned and errno is set to EINVAL.
- *release:* to close the descriptor of an opened device file.

For each part of your implementation, you need to demonstrate that your programs are able to run in a desktop Linux system (e.g. Ubuntu), and an embedded Linux system (i.e., Galileo Gen 2). In order to do this, you will need a makefile that chooses right compilation/build tools and mouse event device, given the target execution environment. This can be done by defining a "test target" variable in *make* command line and using "*ifeq"* conditional directive in a makefile. An example is:

The make command: *make TEST_TARGET=Galileo2*

In makefile:
```
ifeq ($(TEST_TARGET), Galileo2)
        CC = $(TOOLDIR)/i586-poky-linux-gcc
else
        CC = gcc
endif
```

When your programs run in a desktop Linux system, you can utilize few performance monitoring tool to show the execution sequence of the sender and receiver threads. You can use "*trace_cmd*" to collect events from the Linux internal tracer *ftrace*. The traced records can then be viewed via a GUI front end *kernelshark* in a Linux host machine. A report should be compiled to include *kernelshark*'s screen snapshots and to explain the order of execution and preemption of the sender and receiver threads.

Due Date

TBD.

What to Turn in for Grading

- In your Github team repository, you should create a new branch, named "assignment1", where you can save all your work of the assignment, include source code, make and readme files, and a report (in pdf format). In your readme file, the instructions of running your programs should be included.
- Your committed code must be ready in your Github team repository before the due date and will be downloaded on the due date. For the submission committed after the due date, an email notification must be sent to the instructor and the TA. There will be 20 points penalty per day if the submission is late.
- Your team must work on the assignment without any help from other team, and is responsible to the submission committed in Github. No collaboration between teams is allowed, except the open discussion in the forum on Blackboard.
- 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.
- ASU Academic Integrity Policy (http://provost.asu.edu/academicintegrity), and FSE Honor Code (http://engineering.asu.edu/integrity) are strictly enforced and followed. A grade XE will be assigned to any cases of AIP violation.