*LA**nix*** – The LASA Operating System
PQueue Message Lab

Objectives:

- Demonstrate understanding of Queues and Priority Queues by implementing a Priority Queue class.
- Gain experience with Priority Queues in a model of a real world application
- Gain awareness of how Inter Process Communication's are supported in an Operating System (specifically a POSIX compliant OS)

Deliverables:

Basic credit (80%):
- Updated git repository with Pqueue.cpp solution
- Upload solution to BLEND

Content understanding (10%):
- Questionnaire on BLEND submission regarding this write-up

Content application (10%):
- Your choice:
  - Implement additional mqueue.c function in lqueue.cpp
  - Implement additional operations to lq_open.
- Upload source code to both Git and BLEND.

The *LASAnix,* or *LAnix* for short, operating system is LASA's very own Linux derived operating system – with some unique twists. There are many applications of the data structures we learn in Computer Science, but one software package contains nearly all the structure and that software package is an Operating System.

One important service that operating systems provide is Inter Process Communications (IPC). The fundamental purpose of an IPC system is to allow different processes running on a computer to exchange information. A basic requirement is that the IPC mechanisms deliver data in the order it is sent. Most IPC systems are built around the Queue data structure to accomplish this ordered delivery of data. If operating as intended. the First information put In the queue by the sender is the First information Out of the queue at the receiver. - Notice the subtle emphasis on certain letters to achieve the customary acronym '**FIFO**'. No, the customary acronym is NOT FOFI, nor FIFI, nor FumFum – I think those are the sounds a giant makes, not a computer scientist.

***LA**nix* – The LASA Operating System PQueue Message Lab

The LaQueue lab provides a test frame work for the implementation of a priority queue class. The application provides a client and server that communicate with each other over a POSIX style IPC message queue. The server provides a simple toUpper feature that receives test strings, converts them to upper case, and then returns them to the client.

The text information is prioritized and sent with a priority code that the Pqueue class is expected to handle. The effect of prioritizing the information sent to the server is that the data will be processed in an order that is influenced by the timing of the message being placed in the queue and its priority. The result is that the client will print the returned messages in a different order than originally sent.

The student need only complete the PQueue.cpp and PQueue.h source files to enable the basic message communication. The lab does not require any modification to the other source files to achieve basic credit.

This lab is composed of ten(10) source files (.cpp & .h files):

- LaQueue.cpp (main): This launches our server and client “applications”.
- Server.cpp & server.h: is the server application. The server provides a 'toUpper()' operation that echos all messages received as upper case versions of the message. The 'main()' code block is renamed 'server_main()' to avoid conflict with our Lab's main.cpp.
- Client.cpp & client.h: is the client application.
- Lqueue.cpp: is the coding necessary to process the messages through the system and is the code that interacts with the queue data structure. This code is loosely modeled after the code that Linux implements. The most notable difference is that this code runs entirely in user space.
- Lqueue.h: This is the 'mqueue.h' header file Linux provides but with all the 'mq_' prefixes renamed 'lq_' so the POSIX 'mq_open()' functions becomes the *LAnix* 'lq_open()' function and so on...
- Siginfo.h: The Linux signal definitions file straight from Linux. However since this is now part of our user space program and various OS's use different naming conventions (I am looking at you Microsoft), we are keeping this locally in our app so that you may complete the lab on Windows, or Mac, or Linux.
- PQueue.cpp & Pqueue.h: templates for your Priority Queue class implementations.

Code Walkthrough

Let's take a look at the code. There are various points of interest in the lab code that may be helpful to be aware. So pull out the source code and use the following description as a tour guide as you complete this walk through.

LaQueue.cpp

This is the top level of the application. The `main()` function creates the threads that mimic a server application and a client application. Our encounter with the thread library is primarily here. It accomplishes this by calling the POSIX specified `pthread_create()` function for each thread, and then stores the identifying information about the thread provided by the OS.

The format for `pthread_create()` is:

```
int pthread_create(pthread_t *thread, const pthread_attr_t
*attr, void *(*start_routine)(void*), void *arg);
```

where;

`pthread_t *thread`: Location, local to main, for system to put information about the thread when created.
`const pthread_attr_t *attr`: Attributes passed to pthread system describing the requested thread configuration.
`void *(*start_routine)(void*)`: The routine within the thread to begin execution
`void *arg`: The arguments passed into the thread, essentially the argc, argv of '`main(argc, argv)`'

The thread functions are made known to the main code block by the inclusion of '<pthread.h>'. There are many functions available for the management of threads. Our application only needs the functions to create the threads. The child functions will need to execute a corresponding `pthread_exit()` call when it is finished.

The main method also adds a small delay between launching the server thread and the client thread. This allows the server thread to set up the IPC message queue that the client can then access to send messages . The `main()` executes `udelay()` which is a system feature that pauses the currently executing block of code for a period of time defined in nano seconds. The main.cpp code is made aware of the udelay feature by including `<unistd.h>` and `<time.h>`. The `<unistd.h>` provides miscellaneous POSIX symbolic constants and types, and declares miscellaneous functions and `<time.h>` relies on some of these declarations.

Server.cpp and server.h
The server is given the first opportunity to initialize and start running. Its first important task is to open a IPC message queue through which to receive messages from a client.

The lqueue.h header file is included to make server.cpp aware of the messaging functions available to *LAnix* processes. Notice that the include statement uses double quotes rather than angle brackets. The double quotes tell the compiler to look for the header in the local directory first, the angle brackets tell the compiler that the header file is available in the system directories. Since we are providing the message queue library we will keep this locally. The lqueue.h header file we are using is simply the original linux header file with the

'mq_' prefix replaced with 'lq_' for the function prototypes.

The server opens a message queue with the `lq_open()` function

```
lqd_t lq_open (const char *__name, int __oflag, int mode,
lq_attr *attr); .
```

Where

`const char *__name:` - A text name for the channel. In this example we have hard coded a server message name. It is important to note that Linux requires the first character to be a forward slash '/'.

`int __oflag:` Defines the operation of the call. In our case we issue O_CREAT to tell the open function to create the channel.

`int mode:` Defines access priviliges for users, groups, and others and is analogous to the linux file access permissions.

`lq_attr *attr:`. Provides attributes that are applied to the queue such as; message size, max messages for queues,...

Our command sets the queue name and sets the oflag to O_CREAT so that a channel is createded. The function call does apply permission and sets the message queue to ten messages and each message to 255 bytes. However, our lqueue.cpp does not currently manage permissions or attempt to implement the queue attributes. The message size is defaulted to 256 and we does not care how deep the queue gets.

The return value from lq_open is an id number that identifies the queue. The returned value is retained in an array that has positions for up to five(5) queue identifiers. If a '-1' return value indicates an error and the code reports the error and exits.

If the server queue is successfully established, the server goes into a loop looking for messages to process. For some server processes in the operating system the operation is continuous or until something kills the process. Such a main loop might be managed with something like `while(1) { do stuff }`. In our case we will run the server for a limited span of time and our while loop looks like:

```
double TimeToRunInSecs = 0.5;
clock_t c = clock();
while(double(clock()-c)/CLOCKS_PER_SEC < TimeToRunInSecs)
{
```

This uses the clock() function of the system to run for a specified amount of time. We capture the state of the clock before the start of the loop and then within the loop we check the

elapsed time since our initial time stamp we captured in 'c'. We accomplish this with `clock()-c`. The `CLOCKS_PER_SEC` constant is a system supplied value that identifies the clocks per second of a given piece of hardware.

Now we are executing within the body of the loop. The first thing we do is start looking for messages. The `lq_recieve()` function performs this task by obtaining the next message in the queue. This will be the oldest message with the highest priority. The message are expected to be one of two items. Either it is a message with a block of text to be made upper case or it is the name of the clients queue that the server will use to return the converted messages.

To distinguish the types of messages from each other we will take advantage of the fact that linux requires the first character of a queue name to contain a forward slash '/'. When a client queue name is detected, the server attempts to obtain the queue id. Once a client queue name is recieved the server immediately tries to connect to the client queue. The server issues another lq_open command but with a O_WRONLY oflag to indicate that it wants read write access to the queue rather than the O_CREAT oflag used to create the server queue. If the system `lq_open()` can find the channel in the table of established queues, it returns the queue's id which the server can then use to send messages.

Most of the time the information will be text messages to be converted to upper case. Once a block of text is received through the queue, it is converted and sent back through the client's queue. To send the message the server uses the `lq_send()` function. The general form for the send command:

```
int mq_send(mqd_t mqdes, const char *msg_ptr,
                size_t msg_len, unsigned int msg_prio);
```

Where;

`mqd_t mqdes`, - The message queue identifier. We got this from opening the client queue

`const char *msg_ptr`, - A pointer to a buffer containing the message to send

`size_t msg_len`, - The length of the message we are sending.

`unsigned int msg_prio:` - This message's priority. On the return trip they are all the same priority.

At the moment the server assumes the queue will accept the message. If an error is reported from the queue, the error is reported and the server simply moves on to the next message, discarding the failed message.

As implemented the server stays awake for a period of time and processes the messages

from the client. Eventually the timing loop we are using will expire and the processing loop will exit. The last thing the server will do is unlink itself from the message queues with the `mq_unlink(const char *queue_name)` function. The unlink function is pretty straight forward.

Client.cpp

Similar to Server.
The Client application is the reciprocal of the Server. All the same queue operations are used but in differing order. The significant function of the client app is to generate a series of text buffers and send them to the sever, with a different priorities for the messages.

After the clients connects to server's message queue, it creates a a client queue and sends the queue name to the server. The queue name includes a numerical reference to the clients process identifier, or 'pid'. This creates a unique name for this queue. In the case of one client this not a big deal, but it is helpful if we have multiple clients where a unique queue name is important. The process id is obtained by making a system call to getpid() and id number returned by the call is appended to the client queue name.

The client transmits the unique client queue name as the first message. The queue name is required to begin with a forward slash character '/'. This is a requirement of the Linux mqueue library with which we will comply in our lab. The server will use the name we send through the server queue to connect with the client queue. And it is the client queue that the server will return the converted messages.

Lqueue.cpp

Things get interesting here, this code mimics the Linux provided mqueue.c module. It provides the management of the queue data structure and manages the interactions with the processes that access the queues.

An Important aspect of our lqueue version of mqueue is that our queue structure operates in user mode . User mode is a CPU processor state that operating systems use to run applications to prevent them from running amok and crashing the entire system. In user mode there is no direct access by the program to the hardware resources of the system or any other user application. All interaction with the computer's resources is handled through operating system services that operate in Kernel mode.

Operating in Kernel mode allows the OS complete unrestricted access to the hardware resources and can execute any CPU instruction. Only proven trusted code is allowed to run in this mode because failures in kernel mode will crash the entire system and all user

programs. On intel brand CPUs the protective modes are referred to as Rings with Kernel mode as 'Ring 0' and user mode usually running in 'Ring 3'. Detailed discussion can be found at https://manybutfinite.com/post/cpu-rings-privilege-and-protection/

Another important aspect of lqueue.cpp is that it is working with a client and a server process that operate independently. So what happens if each process is trying to work on the same information at the same time? For instance what might happen if one process is trying to access information when the other process is trying delete the same information. Maybe a pointer gets overwritten by a null. Strange results are likely to occur and sometimes a process can be sent off into La La Land, eventually to crash the system. Imagine two cars trying to occupy the same segment of road at the same time, like at a road intersection.

The term 'race condition' is used to describe a situation where two processes need a certain order of operation of code in a segment of a program. The 'race' is for a process to complete a code segment before the other process starts affecting the state of that segment, usually by setting variables that the code depends upon. Such a code segment is described by the term 'critical section'.

Preventing a race condition from occurring is accomplished by limiting access to the critical section to one process at a time. Got any ideas? Back to our car analogy, we handle critical sections such as road intersections with a stoplight. So we need to put a stoplight in our code, got any ideas? One strategy might be to use a flag. We can check the flag and see if it is clear or has a 'green light'. When the flag is clear, the current process sets it to not clear, or 'red light', and proceeds with executing the critical segment. After the segment is completed, the flag is cleared or set back to a 'green light'. That all sounds good, but isn't this just another 'race condition'? Well yes it is. Just a smaller critical section.

The basic principal of controlling mutually exclusive access to critical sections with a flag is fundamentally sound but takes a little more sophistication to accomplish. The POSIX pthread library contains a scheme that we can use. At the beginning of lqueue.cpp a variable myLock of type mutex is declared (`std::mutex mylock`). Whenever a critical section is encountered we can use a lock method (`myLock.lock()`) to prevent interference by other processes. When the process exits the critical section the mutex flag is cleared (`myLock.unlock()`).

If you look at the mq_send and mq_recieve you will notice information is held in buffers local to lqueue. This means all the data that is put in a queue is copied into a buffer created within lqueue. When the information is pulled out of the queue the buffer is released/deleted. Our implementation of lqueue supports non-blocking operation. In other words the queue is checked for data and if nothing is found a length of zero(0) is returned indicating – you guessed it - no data. The alternative is 'blocking' the execution of the process so that it must wait for data before returning. The full mqueue operation supports both blocking and non-blocking operation depending on the configuration flags passed when the queue is created.

Because lqueue creates buffers internal to the messaging system, the server and client don't

have to worry about when they can delete or repurpose a buffer. They have no visibility on when the receiving process removes data so they cannot effectively manage a message buffer as it traverses the messaging system. So the data is copied into a separate buffer at each send or receive request made to lqueue.

The last point we will comment on is that we have kept lqueue minimal compared to it's full fledged linux counter part. The mqueue uses multiple tables to manage the queues and the accessing processes. There is a table to manage the queues and there are tables to manage the processes accesses of the queues. I think there is one other type of table buried in there but I am not quite clear on how it is utilized. To simplify things we have implemented just one table to manage the queues and track links to the queue by use of reference count. The reference count is incremented for each open operation and decremented for each unlink or close operation. The queue is maintained until the reference count goes to zero and it is then deleted from the system.

Building the code

If you were compiling the application with the intent of utilizing the OS provided thread and message services in server.cpp and client.cpp , your compiler directive would look something like:

```
g++ LaQueue.cpp server.cpp client.cpp PQueue.cpp -lpthread -lrt
```

The '-l' tells the compiler and linker to link the thread and librt libraries into the app. 'librt' contains run time features including the IPC messaging functions. All this depend on the operating system, the tool chain, and how you are building the code. The Mac for instance does not provide POSIX messaging so '-lrt' does not exist. Our lqueue module will be the only POSIX compliant IPC on the Mac.

Since we are providing the message queueing system we do not need the '-lrt' when lqueue.cpp is included. The '-lpthread' may or may not be necessary depending on the tool configuration. With our lqueue.cpp module added to the application the compiler directive might look like:

```
g++ LaQueue.cpp server.cpp client.cpp lqueue.cpp PQueue.cpp -lpthread
```

So '-lpthread' may need to be replaced by '-lthread' or possibly nothing at all.

Expected output.

```
Server: Hello, World!
Sever message queue open.
```

```
Client: Hello World:
Client: Token received from server: AAAAAAAAAA
Client: Token received from server: EEEEEEEEEE
Client: Token received from server: IIIIIIIIII
Client: Token received from server: MMMMMMMMM
Client: Token received from server: QQQQQQQQQ
Client: Token received from server: UUUUUUUUUU
Client: Token received from server: YYYYYYYYYY
Client: Token received from server: BBBBBBBBBB
Client: Token received from server: FFFFFFFFFF
Client: Token received from server: JJJJJJJJJJ
Client: Token received from server: NNNNNNNNNN
Client: Token received from server: RRRRRRRRRR
Client: Token received from server: VVVVVVVVVV
Client: Token received from server: ZZZZZZZZZZ
Client: Token received from server: CCCCCCCCCC
Client: Token received from server: GGGGGGGGGG
Client: Token received from server: KKKKKKKKKK
Client: Token received from server: OOOOOOOOOO
Client: Token received from server: SSSSSSSSSS
Client: Token received from server: WWWWWWWWWW
Client: Token received from server: DDDDDDDDDD
Client: Token received from server: HHHHHHHHHH
Client: Token received from server: LLLLLLLLLL
Client: Token received from server: PPPPPPPPPP
Client: Token received from server: TTTTTTTTTT
Client: Token received from server: XXXXXXXXXX
Client: bye
Server: exit.
```