```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#define SH_Q_NAME            "/myIPCSharedQ"
#define SH_Q_QSIZE           8

#define MUTEX_NAME           "/mySharedRWMutex"
#define CONDVAR_NAME   "/mySharedRWCondVar"

#define LOG(format, ...) printf("[PID:%d] ",getpid()); printf(format,
##__VA_ARGS__)

typedef struct{
    char buffer[20];
    size_t bufferLen;
    uint8_t usrLed_onoff:1;
}payload_t;

int main()
{
      struct mq_attr mysharedQ_attr;
      mysharedQ_attr.mq_maxmsg = SH_Q_QSIZE;
      mysharedQ_attr.mq_msgsize = sizeof(payload_t);

      mqd_t mySharedQ = mq_open(SH_Q_NAME, O_CREAT | O_RDWR, 0666,
&mysharedQ_attr);

      if(mySharedQ == (mqd_t)-1)
      {
            LOG("[ERROR] QUEUE OPEN ERROR.: %s\n",strerror(errno));
            return -1;
      }

    char *payload_cptr;
    payload_t payloadRecv = {0};
    payload_cptr = (char*)&payloadRecv;

      int ret = mq_receive(mySharedQ, payload_cptr,
sizeof(payloadRecv),0);
      if(ret == -1)
      {
            LOG("[ERROR] Q Send error: %s\n",strerror(errno));
            return -1;
      }
      LOG("[INFO] Message recd size: %d\n",ret);
      //payloadptr = (payload_t*)readbuf;
    LOG("[INFO] Message Dequeued\n{Message: %s\nMessageLen: %d\nUSRLED:
%d}\n",payloadRecv.buffer,payloadRecv.bufferLen,payloadRecv.usrLed_onoff)
;
```

```
    const char* msg = "Hello from Process2";
    payload_t payloadSend = {0};
    payload_cptr = (char*)&payloadSend;

    memmove(payloadSend.buffer,msg,strlen(msg));
    payloadSend.bufferLen = strlen(payloadSend.buffer);
    payloadSend.usrLed_onoff = 1;

      ret = mq_send(mySharedQ, payload_cptr, sizeof(payloadSend),0);
      if(ret == -1)
      {
            LOG("[ERROR] Q Send error: %s\n",strerror(errno));
            return -1;
      }

      LOG("[INFO] Message Queued\n{Message: %s\nMessageLen: %d\nUSRLED:
%d}\n",payloadSend.buffer,payloadSend.bufferLen,payloadSend.usrLed_onoff)
;


      /*Closing the Q. Process 1 will destroy the queue */
      mq_close(mySharedQ);

      LOG("[INFO] QUEUE CLOSED\n");
      return 0;
}#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#define SH_Q_NAME            "/myIPCSharedQ"
#define SH_Q_QSIZE           8

#define MUTEX_NAME           "/mySharedRWMutex"
#define CONDVAR_NAME   "/mySharedRWCondVar"

#define LOG(format, ...) printf("[PID:%d] ",getpid()); printf(format,
##__VA_ARGS__)

typedef struct{
    char buffer[20];
    size_t bufferLen;
    uint8_t usrLed_onoff:1;
}payload_t;

int main()
{
      struct mq_attr mysharedQ_attr;
      mysharedQ_attr.mq_maxmsg = SH_Q_QSIZE;
      mysharedQ_attr.mq_msgsize = sizeof(payload_t);

      mqd_t mySharedQ = mq_open(SH_Q_NAME, O_CREAT | O_RDWR, 0666,
&mysharedQ_attr);
```

```
      if(mySharedQ == (mqd_t)-1)
      {
            LOG("[ERROR] QUEUE OPEN ERROR.: %s\n",strerror(errno));
            return -1;
      }

      const char* msg = "Hello from Process1";
    char *payload_cptr;
    payload_t payloadSend = {0};
    payload_cptr = (char*)&payloadSend;

    memmove(payloadSend.buffer,msg,strlen(msg));
    payloadSend.bufferLen = strlen(payloadSend.buffer);
    payloadSend.usrLed_onoff = 1;

      int ret = mq_send(mySharedQ, payload_cptr, sizeof(payloadSend),0);
      if(ret == -1)
      {
            LOG("[ERROR] Q Send error: %s\n",strerror(errno));
            return -1;
      }

      LOG("[INFO] Message Queued\n{Message: %s\nMessageLen: %d\nUSRLED:
%d}\n",payloadSend.buffer,payloadSend.bufferLen,payloadSend.usrLed_onoff)
;

      LOG("[INFO] Will wait for Process 2 to enqueue some message\n");

    payload_t payloadRecv = {0};
    payload_cptr = (char*)&payloadRecv;

      ret = mq_receive(mySharedQ, payload_cptr, sizeof(payloadRecv),0);
      if(ret == -1)
      {
            LOG("[ERROR] Q Send error: %s\n",strerror(errno));
            return -1;
      }
      LOG("[INFO] Message recd size: %d\n",ret);
      //payloadptr = (payload_t*)readbuf;
    LOG("[INFO] Message Dequeued\n{Message: %s\nMessageLen: %d\nUSRLED:
%d}\n",payloadRecv.buffer,payloadRecv.bufferLen,payloadRecv.usrLed_onoff)
;

      mq_unlink(SH_Q_NAME);

      LOG("[INFO] QUEUE DESTROYED\n");

      return 0;
}

.PHONY:default
default:
      @echo "Build Started"
      gcc -g3 process1.c -o proc1 -lrt
      gcc -g3 process2.c -o proc2 -lrt
      @echo "Build Completed"
```

```
.PHONY:clean
clean:
	rm -rf proc1 proc2
#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */

#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <pthread.h>
#include <semaphore.h>

#define SH_MEM_NAME    "/MY_SH_MEM"
#define SH_MEM_SIZE sizeof(payload_t)

#define SEM_NAME "/sharedMemSemaphore"

#define LOG(format, ...) printf("[PID:%d] ",getpid()); printf(format,
##__VA_ARGS__)

typedef struct{
    char buffer[20];
    size_t bufferLen;
    uint8_t usrLed_onoff:1;
}payload_t;


int main()
{
	LOG("[INFO] Starting the process 2\n");
	int shmem_fd = shm_open(SH_MEM_NAME, O_CREAT | O_RDWR, 0666);
	if(shmem_fd < 0 )
	{
		LOG("[ERROR] Cannot open Shared Mem: %s",strerror(errno));
		return -1;
	}

	void *shared_mem = mmap(NULL, SH_MEM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED , shmem_fd, 0);
	if(shared_mem == (void*)-1)
	{
		LOG("[ERROR] mmap error: %s\n", strerror(errno));
		return -1;
	}

	/* Creating a semaphore to sync the reads and writes between 2
processes */
	sem_t *sem = sem_open(SEM_NAME, O_CREAT, 0666, 0);
	if(SEM_FAILED == sem)
	{
		LOG("[ERROR] Sem open Failed:%s\n",strerror(errno));
		return -1;
	}
```

```
      /* Waiting for the process 1 to post the sem after writing data to
the shared mem */
      sem_wait(sem);

      payload_t payloadRecv = {0};
    char *payload_cptr = (char*)&payloadRecv;

    memcpy(payload_cptr,(char*)shared_mem,SH_MEM_SIZE);

    LOG("[INFO] Message From Proc 1 through Shared Mem\n{Message:
%s\nMessageLen: %d\nUSRLED:
%d}\n",payloadRecv.buffer,payloadRecv.bufferLen,payloadRecv.usrLed_onoff)
;

    const char* msg = "Hello from Process2";
    payload_t payloadSend = {0};
    payload_cptr = (char*)&payloadSend;

    memmove(payloadSend.buffer,msg,strlen(msg));
    payloadSend.bufferLen = strlen(payloadSend.buffer);
    payloadSend.usrLed_onoff = 1;

    /* Copy the contents of the payload into the share memory */
    memcpy((char*)shared_mem, payload_cptr, SH_MEM_SIZE);

    /*Indicating the process 1 that the data has been written for Process
1 eyes only */
    sem_post(sem);

      /*Closing the shared memory handle*/
      int ret = close(shmem_fd);
      if(ret < 0)
      {
            LOG("[ERROR] Cannot close Shared Mem: %s",strerror(errno));
            return -1;
      }

      return 0;
}#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */

#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <pthread.h>
#include <semaphore.h>

#define SH_MEM_NAME    "/MY_SH_MEM"
#define SH_MEM_SIZE sizeof(payload_t)

#define SEM_NAME "/sharedMemSemaphore"

#define LOG(format, ...) printf("[PID:%d] ",getpid()); printf(format,
##__VA_ARGS__)
```

```
typedef struct{
    char buffer[20];
    size_t bufferLen;
    uint8_t usrLed_onoff:1;
}payload_t;

int main()
{
      LOG("[INFO] Starting the process 1\n");
      int shmem_fd = shm_open(SH_MEM_NAME, O_CREAT | O_RDWR, 0666);
      if(shmem_fd < 0)
      {
            LOG("[ERROR] Cannot open Shared Mem: %s",strerror(errno));
            return -1;
      }

      int ret = ftruncate(shmem_fd, SH_MEM_SIZE);
      if(ret < 0)
      {
            LOG("[ERROR] ftruncate on share mem: %s",strerror(errno));
            return -1;
      }

      void *shared_mem = mmap(NULL, SH_MEM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED , shmem_fd, 0);
      if(shared_mem == (void*)-1)
      {
            LOG("[ERROR] mmap error: %s\n",strerror(errno));
            return -1;
      }

      /* Creating a semaphore to sync the reads and writes between 2
processes */
      sem_t *sem = sem_open(SEM_NAME, O_CREAT, 0666, 0);
      if(SEM_FAILED == sem)
      {
            LOG("[ERROR] Sem open Failed:%s\n",strerror(errno));
            return -1;
      }

      const char* msg = "Hello from Process1";
    char *payload_cptr;
    payload_t payloadSend = {0};
    payload_cptr = (char*)&payloadSend;

    memmove(payloadSend.buffer,msg,strlen(msg));
    payloadSend.bufferLen = strlen(payloadSend.buffer);
    payloadSend.usrLed_onoff = 1;

    /* Copy the contents of the payload into the share memory */
    memcpy((char*)shared_mem, payload_cptr, SH_MEM_SIZE);

    /*Indicating the process 2 that the data has been written for Process
2 eyes only */
    sem_post(sem);

      /* Waiting for the process 2 to post the sem after writing data to
the shared mem */
```

```
    sem_wait(sem);

    payload_t payloadRecv = {0};
    payload_cptr = (char*)&payloadRecv;

    memcpy(payload_cptr,(char*)shared_mem,SH_MEM_SIZE);

    LOG("[INFO] Message From Process 2 through Shared Mem\n{Message:
%s\nMessageLen: %d\nUSRLED:
%d}\n",payloadRecv.buffer,payloadRecv.bufferLen,payloadRecv.usrLed_onoff)
;


      /*Destroying the shared memory */
      ret = shm_unlink(SH_MEM_NAME);
      if(ret < 0)
      {
            LOG("[ERROR] Cannot destroy Shared Mem: %s",strerror(errno));
            return -1;
      }

      return 0;
}

.PHONY:default
default:
      @echo "Build Started"
      gcc -g3 process1.c -o proc1 -lpthread -lrt
      gcc -g3 process2.c -o proc2 -lpthread -lrt
      @echo "Build Completed"

.PHONY:clean
clean:
      rm -rf proc1 proc2
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

typedef struct{
    char buffer[20];
    size_t bufferLen;
    uint8_t usrLed_onoff:1;
}payload_t;

#define LOG(format, ...) printf("[PID:%d] ",getpid()); printf(format,
##__VA_ARGS__)

int main()
{
    // We use two pipes
    // First pipe to send input string from parent
    // Second pipe to send concatenated string from child

    int Par_to_Ch[2];  // Used to store two ends of first pipe
```

```
    int Ch_to_Par[2];  // Used to store two ends of second pipe

    pid_t p;

    if (pipe(Par_to_Ch)==-1)
    {
        LOG("[ERROR] Pipe call error\n");
        return 1;
    }
    if (pipe(Ch_to_Par)==-1)
    {
        LOG("[ERROR] Pipe call error\n");
        return 1;
    }

    p = fork();

    if (p < 0)
    {
        LOG("[ERROR] Fork error\n");
        return 1;
    }

    // Parent process
    else if (p > 0)
    {
        LOG("PARENT\n");
        char concat_str[100];
        const char* msg = "Hello from Parent";
        char *payload_cptr;
        payload_t payloadSend = {0};
        payload_cptr = (char*)&payloadSend;

        memcpy(payloadSend.buffer,msg,strlen(msg)+1);
        payloadSend.bufferLen = strlen(payloadSend.buffer);
        payloadSend.usrLed_onoff = 1;

        close(Par_to_Ch[0]);

        write(Par_to_Ch[1], payload_cptr, sizeof(payloadSend));
        LOG("[INFO] Message sent to child from parent\n");
        close(Par_to_Ch[1]);

        /* Wait for child to send a string */
        wait(NULL);

        close(Ch_to_Par[1]);

        char readbuf[sizeof(payload_t)] = {0};
        payload_t *payloadptr;

        read(Ch_to_Par[0], readbuf, sizeof(payload_t));

        payloadptr = (payload_t*)readbuf;
        LOG("[INFO] Message Recvd\n{Message: %s\nMessageLen: %d\nUSRLED:
%d}\n",payloadptr->buffer,payloadptr->bufferLen,payloadptr-
>usrLed_onoff);
```

```
        close(Ch_to_Par[0]);

        exit(0);
    }
    /* child process */
    else
    {
        LOG("CHILD\n");
        close(Par_to_Ch[1]);

        char readbuf[sizeof(payload_t)] = {0};
        payload_t *payloadptr;

        read(Par_to_Ch[0], readbuf, sizeof(payload_t));

        payloadptr = (payload_t*)readbuf;
        LOG("[INFO] Message Recvd\n{Message: %s\nMessageLen: %d\nUSRLED:
%d}\n",payloadptr->buffer,payloadptr->bufferLen,payloadptr-
>usrLed_onoff);

        close(Par_to_Ch[0]);

        close(Ch_to_Par[0]);

        const char* msg = "Hello from Child";
        char *payload_cptr;
        payload_t payloadSend = {0};
        payload_cptr = (char*)&payloadSend;

        memcpy(payloadSend.buffer,msg,strlen(msg)+1);
        payloadSend.bufferLen = strlen(payloadSend.buffer);
        payloadSend.usrLed_onoff = 0;

        write(Ch_to_Par[1], payload_cptr, sizeof(payloadSend));
        LOG("[INFO] Message sent to parent from child\n");

        close(Ch_to_Par[1]);

        exit(0);
    }
}

.PHONY:default
default:
      @echo "Build Started"
      gcc -g3 pipe_demo.c -o pipe_demo
      @echo "Build Completed"

.PHONY:clean
clean:
      rm -rf pipe_demo
#include <sys/socket.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
#define PORT     2000
#define IP

#define LOG(format, ...) printf(format, ##__VA_ARGS__)

typedef struct{
    char buffer[20];
    size_t bufferLen;
    uint8_t usrLed_onoff:1;
}payload_t;


int main()
{
      int server_socket, accepted_socket, option = 1;
      struct sockaddr_in addr, peer_addr;
      int addrLen = sizeof(peer_addr);
      payload_t payload_recvd = {0};

      if((server_socket = socket(AF_INET,SOCK_STREAM,0)) == 0)
      {
            LOG("[ERROR] Socket Creation\n");
            return 1;
      }

      LOG("[INFO] Socket Created\n");

      if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &(option),
sizeof(option)))
    {
        LOG("[ERROR] Cannot Set socket options\n");
        return 1;
    }
      /*Setting up the sockaddr_in structure */
      addr.sin_family = AF_INET;
      /* Change the below address to our IP addr */
      //addr.sin_addr.s_addr = inet_addr("192.168.1.238");//INADDR_ANY;

      addr.sin_addr.s_addr = INADDR_ANY;//Using local loopback
      addr.sin_port = htons(PORT);

      if((bind(server_socket,(struct sockaddr*)&addr, sizeof(addr))) < 0)
      {
            LOG("[ERROR] Cannot bind the socket\n");
            return 1;
      }

      LOG("[INFO] Socket binded\n");

      if(listen(server_socket,5) < 0)
      {
            LOG("[ERROR] Cannot listen\n");
            return 1;
      }

      //while(1)
      //{
```

```
            accepted_socket = accept(server_socket, (struct
sockaddr*)&peer_addr,(socklen_t*)&addrLen);
            if(accepted_socket < 0)
            {
                  LOG("[ERROR] Cannot accept\n");
      //          continue;
                  return 1;
            }

            char peer_IP[20] = {0};
            LOG("[INFO] Peer Addr: %s\n",inet_ntop(AF_INET,
&peer_addr.sin_addr, peer_IP, sizeof(peer_IP)));

            char readBuffer[1024] = {0};
            int bytesRead;
            size_t payloadLen = 0;
            bytesRead = read(accepted_socket, &payloadLen,
sizeof(size_t));

            if(bytesRead == sizeof(size_t))
            {
                  LOG("[INFO] Size of incoming payload: %d\n",payloadLen);
            }
            else
            {
                  LOG("[ERROR] Invalid data\n");
                  return 1;
            }
            int i = 0;
            while((bytesRead = read(accepted_socket, readBuffer+i, 1024))
< payloadLen)
            {
                  LOG("[INFO] Number of bytes recvd: %d\n",bytesRead);
                  i+=bytesRead;
            }

            payload_t *payloadptr= (payload_t*)readBuffer;
            LOG("[INFO] Message Recvd\nMessage: %s\nMessageLen:
%d\nUSRLED: %d\n",payloadptr->buffer,payloadptr->bufferLen,payloadptr-
>usrLed_onoff);
      //}

            send(accepted_socket , "ACK" , 4, 0);
            close(accepted_socket);

      return 0;
}#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <arpa/inet.h>

#define PORT 2000
#define IP "127.0.0.1"
//#define IP "192.168.1.238"
#define LOG(format, ...) printf(format, ##__VA_ARGS__)
```

```
typedef struct{

    char buffer[20];
    size_t bufferLen;
    uint8_t usrLed_onoff:1;
}payload_t;

int main()
{
    struct sockaddr_in addr, server_addr = {0};
    int client_socket = 0;
    const char* msg = "Hello from Client";
    char *payload_ptr;
    payload_t payloadSend;// = {0};
    payload_ptr = (char*)&payloadSend;

    memcpy(payloadSend.buffer,msg,strlen(msg)+1);
    payloadSend.bufferLen = strlen(payloadSend.buffer);
    payloadSend.usrLed_onoff = 1;

    if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        LOG("[ERROR] Socket creation\n");
        return -1;
    }

    LOG("[INFO] Socket Created\n");

    //memset(&server_addr, 0, sizeof(server_addr));

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);

    /* We need this to convert the IP ADDR in proper format */
    if(inet_pton(AF_INET, IP, &server_addr.sin_addr)<=0)
    {
        LOG("[ERROR] Invalid address\n");
        return -1;
    }

    if (connect(client_socket, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0)
    {
        LOG("[ERROR] Connection Failed \n");
        return -1;
    }

    /*First sending the size of the incoming payload */
    size_t sizeofPayload = sizeof(payloadSend);
    int bytesSent = send(client_socket,&sizeofPayload,sizeof(size_t), 0);
    LOG("[INFO] Sent payload size\n");

    /*Sending the actual payload */
    bytesSent = send(client_socket , (char*)&payloadSend ,
sizeof(payloadSend), 0 );
```

```
    if(bytesSent < sizeof(payloadSend))
    {
        LOG("[ERROR] Cannot send complete data\n");
        return 1;
    }

    LOG("[INFO] Number of bytes send: %d\n",bytesSent);
    LOG("[INFO] Message sent\nMessage: %s\nMessageLen: %d\nUSRLED:
%d\n",payloadSend.buffer,payloadSend.bufferLen,payloadSend.usrLed_onoff);

    char ack[4] = {0};
    read(client_socket, ack, 4);
    LOG("[INFO] return: %s\n",ack);

    close(client_socket);
    return 0;
}

.PHONY:default
default:
      @echo "Build Started"
      gcc -g3 client.c -o client
      gcc -g3 server.c -o server
      @echo "Build Completed"

.PHONY:clean
clean:
      rm -rf client server
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/pid.h>


#define getStatusString(state) ((state > 0) ? "Stopped" : ((state == 0) ?
"Runnable" : ((state == -1) ? "Unrunnable" : "Unknown")))

#define getChildrenCount(child_taskStruct)          \
({                                                        \
      static unsigned int child_count = 0;          \
      struct list_head *list_itr   ;                      \
      list_for_each(list_itr,child_taskStruct)      \
      {                                                   \
            child_count++;                                \
      }                                                   \
      child_count;                                  \
})


static int process_id = -1;

module_param(process_id,int,S_IRUGO | S_IWUSR);

int __init gunjModule_proctree_init(void)
{
      struct task_struct *task;
```

```
      printk(KERN_INFO "Initializing Process tree example Module.
Function %s\n",__FUNCTION__);

      if(-1 == process_id)
      {
            printk(KERN_INFO "Got no process_id as parameter. Taking
current process.\n");
            task = current;
      }
      else
      {
            struct pid *procid_struct = find_get_pid(process_id);
            task = pid_task(procid_struct, PIDTYPE_PID);
      }

      /*
      Thread Name
      Process ID
      Process Status
      Number of children
      Nice value
      */
    printk(KERN_INFO "Process got as parameter: %s, PID: %d, State: %s,
#Children: %u, Nice: %d", task->comm, task->pid, getStatusString(task-
>state), getChildrenCount(&task->children),  task_nice(task));
    do
    {
        task = task->parent;
        printk(KERN_INFO "Parent process: %s, PID: %d, State: %s,
#Children: %u, Nice: %d", task->comm, task->pid, getStatusString(task-
>state), getChildrenCount(&task->children),  task_nice(task));

    }while(0 != task->pid);

      return 0;
}


void __exit gunjModule_proctree_exit(void)
{
      printk(KERN_INFO "Exiting Process tree example Module. Function
%s\n",__FUNCTION__);
}


module_init(gunjModule_proctree_init);
module_exit(gunjModule_proctree_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Gunj Manseta");
MODULE_DESCRIPTION("Module accepts a process id and it prints the details
of all its fore-father processes traversing up the evolution of the
current process lineage up until the big bang.");
MODULE_ALIAS("Gunj_processstree_Module");
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
```

```
#include <linux/kernel.h>
#include <linux/timer.h>
#include <linux/kthread.h>
#include <linux/delay.h>
#include <linux/kfifo.h>
#include <linux/sched.h>

#define MY_KFIFO_NAME  mykfifo
#define MY_KFIFO_NAME_P      &mykfifo

/*Should be a power of 2 */
#define SIZE_SHIFT           4
#define MY_KFIFO_SIZE  (1<<SIZE_SHIFT)


static DEFINE_MUTEX(fifo_lock);

static int dataProducedCount = 0;
static int dataConsumedCount = 0;

struct task_struct *producer_task;
struct task_struct *consumer_task;

static DECLARE_KFIFO(MY_KFIFO_NAME,struct task_struct*,MY_KFIFO_SIZE);


static unsigned long stimeInterval= 5;
module_param(stimeInterval, ulong, S_IRUGO | S_IWUSR);

int producer_callback(void *params)
{
      printk(KERN_INFO "From %s\n",__FUNCTION__);

      while(!kthread_should_stop())
      {
            /* Lock the mutex*/
            if (mutex_lock_interruptible(&fifo_lock))
            {
                  printk(KERN_ERR "Cannot get the lock\n");
                  //return -1;
                  return -ERESTARTSYS;
            }
            /* Push the data into kfifo*/
            if(0 == kfifo_put(MY_KFIFO_NAME_P, current))
                  printk(KERN_INFO "KFIFO FULL\n");
            else
            {
                  //printk(KERN_INFO "Process pushed id: %d\n",current-
>pid);
            }

            /* Unlock the mutex*/
            mutex_unlock(&fifo_lock);

            /* Signal the condition variable */

            dataProducedCount++;
            ssleep(stimeInterval);
```

```
	}

	printk(KERN_INFO "%s is terminated\n",__FUNCTION__);

	return dataProducedCount;
}

int consumer_callback(void *params)
{
	struct task_struct *fifoData;
	printk(KERN_INFO "From %s\n",__FUNCTION__);

	while(!kthread_should_stop())
	{
		/* Lock the mutex*/
		if (mutex_lock_interruptible(&fifo_lock))
		{
			printk(KERN_ERR "Cannot get the lock\n");
			//return -1;
			return -ERESTARTSYS;
		}

		/* Wait for the condition variable */

		/* Pop the data from kfifo*/
		if(0 == kfifo_get(MY_KFIFO_NAME_P, &fifoData))
		{
			//printk(KERN_INFO "KFIFO EMPTY\n");
		}
		else
		{
			/* Process Id and Vruntime */
			printk(KERN_INFO "Previous Process ID: %d, Vruntime:
%llu\n",list_prev_entry(fifoData, tasks)->pid, list_prev_entry(fifoData,
tasks)->se.vruntime);
			printk(KERN_INFO "Current Process ID: %d, Vruntime:
%llu\n",fifoData->pid, fifoData->se.vruntime);
			printk(KERN_INFO "Next Process ID: %d, Vruntime:
%llu\n",list_next_entry(fifoData, tasks)->pid, list_next_entry(fifoData,
tasks)->se.vruntime);
			dataConsumedCount++;
		}

		/* Unlock the mutex*/
		mutex_unlock(&fifo_lock);


	}

	printk(KERN_INFO "%s is terminated\n",__FUNCTION__);

	return dataConsumedCount;
}

int __init gunjModule_kfifoEX_init(void)
{
	printk(KERN_INFO "Initializing kthread kfifo example Module.
Function %s\n",__FUNCTION__);
```

```
	/* Init a kfifo */
	INIT_KFIFO(mykfifo);

	/* Create two threads */
	producer_task = kthread_run(producer_callback,NULL,"Producer
Task");
	if(IS_ERR(producer_task))
	{
		printk(KERN_ERR "Producer Thread run failed.\n");
		return -1;
	}

	consumer_task = kthread_run(consumer_callback,NULL,"Consumer
Task");
	if(IS_ERR(consumer_task))
	{
		int ret;
		printk(KERN_ERR "Consumer thread run failed.\n");
		ret = kthread_stop(producer_task);
		if(-1 != ret)
		{
			printk(KERN_INFO "Producer Thread has stopped with
%d\n",ret);
		}
		return -1;
	}

	/* Everything went as expected */
	return 0;
}


void __exit gunjModule_kfifoEX_exit(void)
{
	/* Delete the kfifo */

	/* Stop the kthreads created */
	int ret = kthread_stop(producer_task);
	if(-1 != ret)
	{
		printk(KERN_INFO "Producer thread has stopped. Data Produced
Count:%d\n",ret);
	}
	else printk(KERN_ERR "Error in Producer Thread");

	ret = kthread_stop(consumer_task);
	if(-1 != ret)
	{
		printk(KERN_INFO "Consumer thread has stopped. Data Consumed
Count:%d\n",ret);
	}
	else printk(KERN_ERR "Error in Consumer Thread");

	printk(KERN_INFO "Exiting Kthread kfifo example Module. Function
%s\n",__FUNCTION__);
}
```

```
module_init(gunjModule_kfifoEX_init);
module_exit(gunjModule_kfifoEX_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Gunj Manseta");
MODULE_DESCRIPTION("Module having two threads. Thread1(Producer) passes
information of the currently scheduled process to the Thread2(Consumer)
using a kfifo.");
MODULE_ALIAS("Gunj_Kthread_ex_Module");
KERNEL_DIR  ?= /lib/modules/$(shell uname -r)/build
PWD         = $(shell pwd)

obj-m := module_kthread_comm.o
obj-m += module_proctree.o


default:
      make -C $(KERNEL_DIR) M=$(PWD) modules


clean:
      make -C $(KERNEL_DIR) M=$(PWD) clean
```