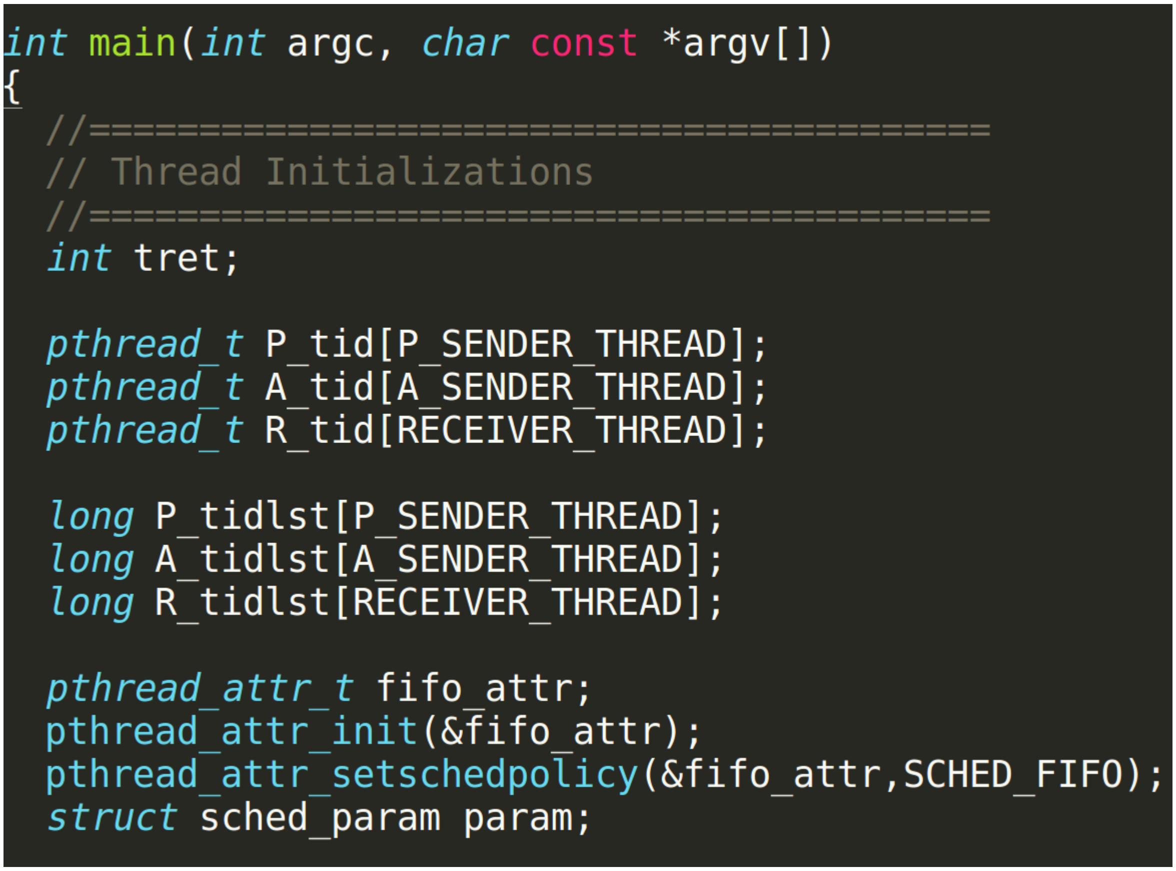
//##

// Project Title : Shared Message Queues in User Space

// Created by : Sarvesh Patil & Nagarjun Chinnari

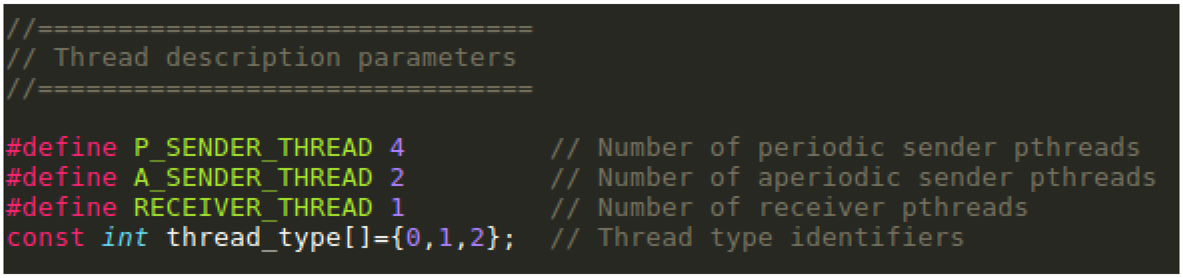
// Date : 25 September 2017
//##

MAIN FUNCTION :



```
int main(int argc, char const *argv[])
{
  //===========================================
  // Thread Initializations
  //===========================================
  int tret;

  pthread_t P_tid[P_SENDER_THREAD];
  pthread_t A_tid[A_SENDER_THREAD];
  pthread_t R_tid[RECEIVER_THREAD];

  long P_tidlst[P_SENDER_THREAD];
  long A_tidlst[A_SENDER_THREAD];
  long R_tidlst[RECEIVER_THREAD];

  pthread_attr_t fifo_attr;
  pthread_attr_init(&fifo_attr);
  pthread_attr_setschedpolicy(&fifo_attr,SCHED_FIFO);
  struct sched_param param;
```

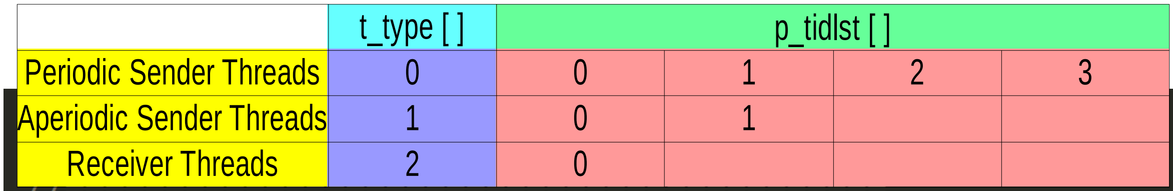
“P_tid[P_SENDER_THREAD]” This is a array which holds thread ids of periodic sender threads.
“A_tid[A_SENDER_THREAD]” This is a array which holds thread ids of the aperiodic sender threads.
“R_tid[RECEIVER_THREAD]” This is a array which holds thread ids of periodic receiver threads.



```
//================================
// Thread description parameters
//================================

#define P_SENDER_THREAD 4          // Number of periodic sender pthreads
#define A_SENDER_THREAD 2          // Number of aperiodic sender pthreads
#define RECEIVER_THREAD 1          // Number of receiver pthreads
const int thread_type[]={0,1,2};   // Thread type identifiers
```

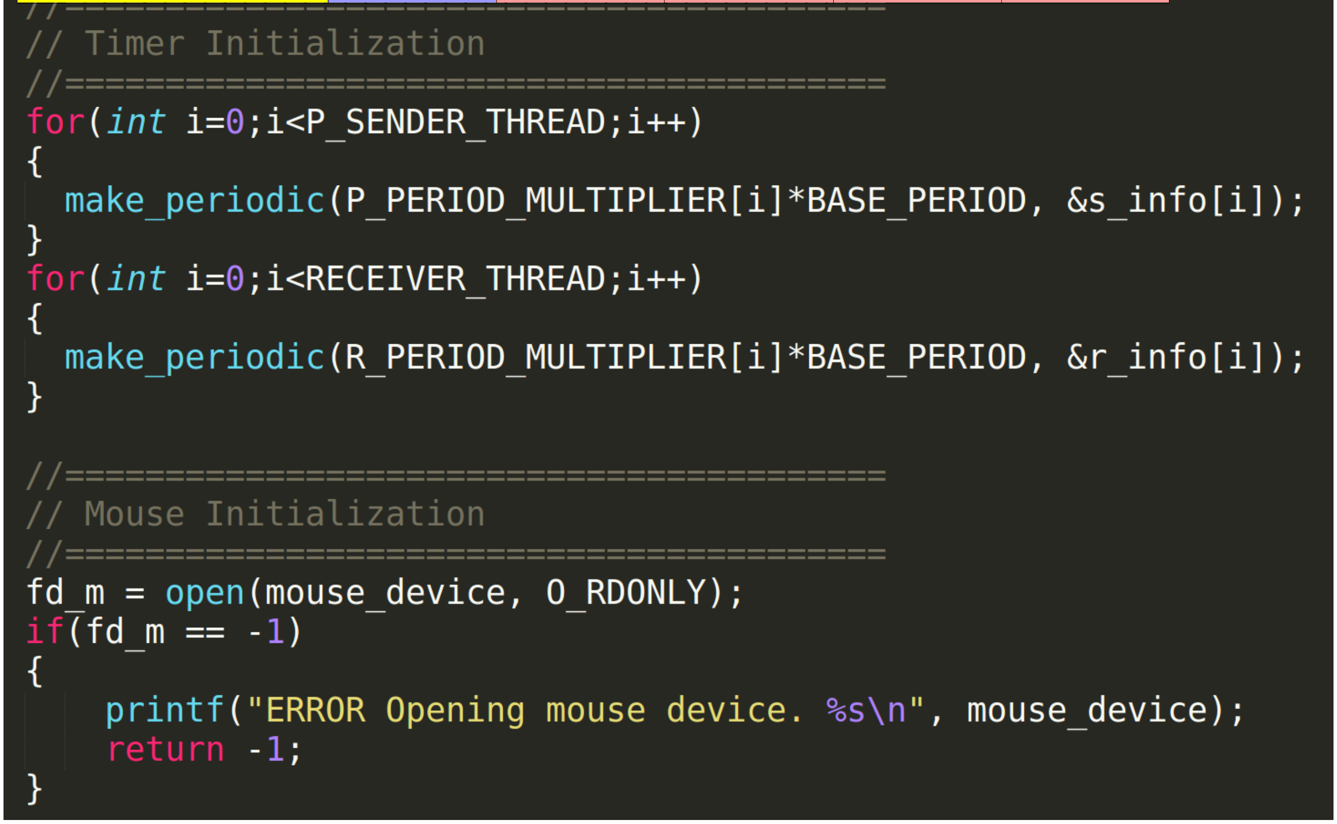
In global scope, thread type[] array is used that distinguishes type of pthread.
This combined with p_tidlist[] array gives unique combination to identify every pthread in code.

e.g. 2nd periodic thread in system will have ID 01.



	t_type [ ]	p_tidlst [ ]			
Periodic Sender Threads	0	0	1	2	3
Aperiodic Sender Threads	1	0	1		
Receiver Threads	2	0			

Semaphore,
Timer &



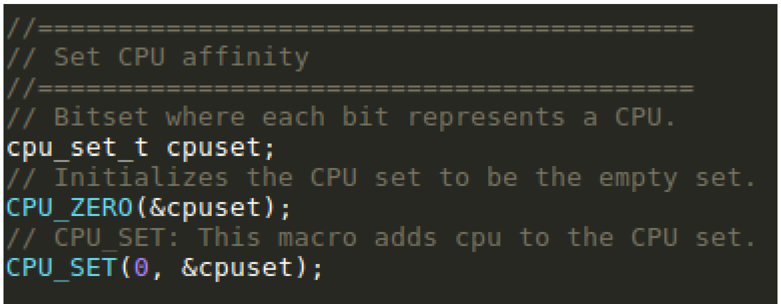
```
//==========================================
// Timer Initialization
//==========================================
for(int i=0;i<P_SENDER_THREAD;i++)
{
  make_periodic(P_PERIOD_MULTIPLIER[i]*BASE_PERIOD, &s_info[i]);
}
for(int i=0;i<RECEIVER_THREAD;i++)
{
  make_periodic(R_PERIOD_MULTIPLIER[i]*BASE_PERIOD, &r_info[i]);
}

//==========================================
// Mouse Initialization
//==========================================
fd_m = open(mouse_device, O_RDONLY);
if(fd_m == -1)
{
    printf("ERROR Opening mouse device. %s\n", mouse_device);
    return -1;
}
```

Mouse are initialized in this sections.

Semaphore is initialized with value flag set to 1. (sem_wait will decrement it to 0 & sem_post will increment back to 1)

Timer is initialized with corresponding periods of each thread.

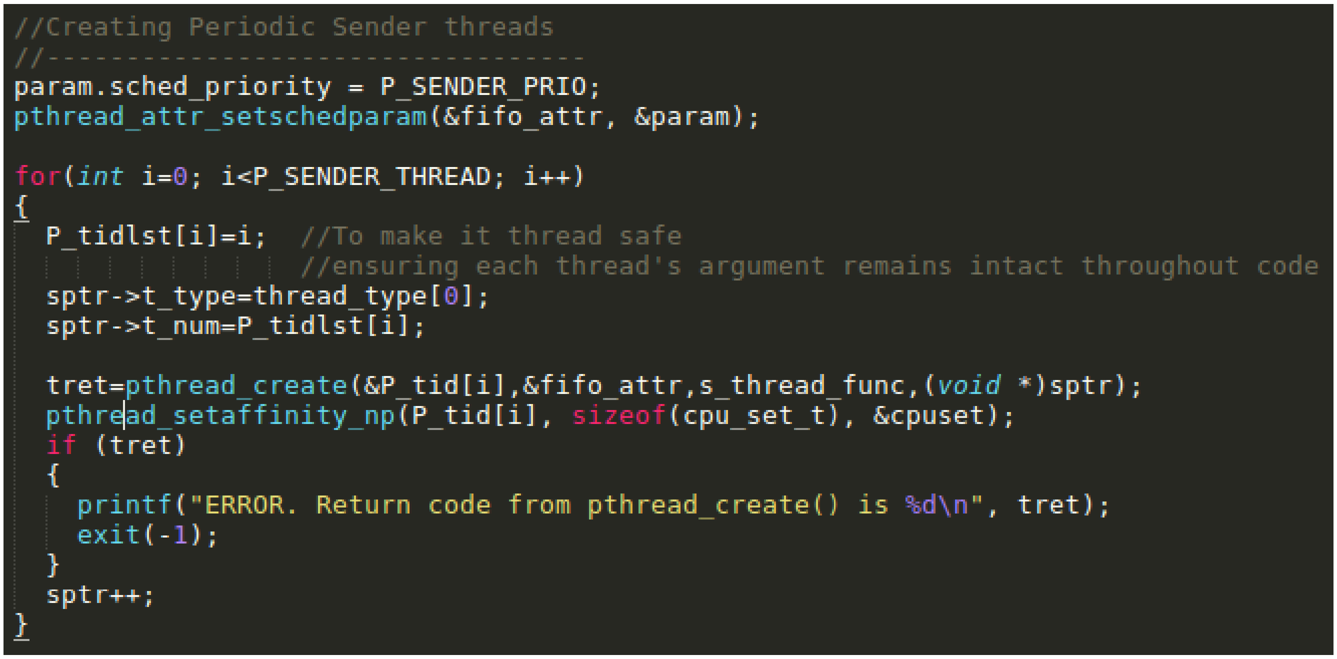
Mouse is initialized with open() function.



```
//========================================
// Set CPU affinity
//========================================
// Bitset where each bit represents a CPU.
cpu_set_t cpuset;
// Initializes the CPU set to be the empty set.
CPU_ZERO(&cpuset);
// CPU_SET: This macro adds cpu to the CPU set.
CPU_SET(0, &cpuset);
```

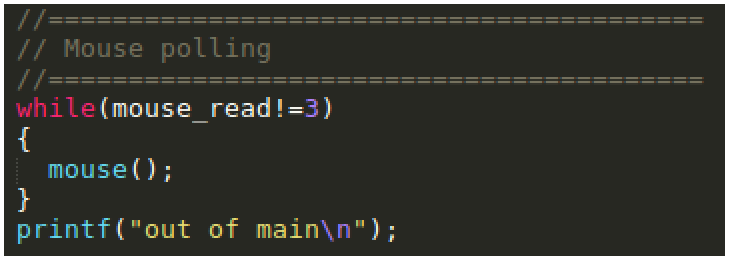
CPU affinity functions set cpuset to core 0.

Periodic, Aperiodic & Receiver threads are created as many as defined in respective macro.
e.g. below is for loop creating 4 periodic sender threads. Thread IDs are stored in P_tid[] array.



```
//Creating Periodic Sender threads
//----------------------------------
param.sched_priority = P_SENDER_PRIO;
pthread_attr_setschedparam(&fifo_attr, &param);

for(int i=0; i<P_SENDER_THREAD; i++)
{
  P_tidlst[i]=i;  //To make it thread safe
                  //ensuring each thread's argument remains intact throughout code
  sptr->t_type=thread_type[0];
  sptr->t_num=P_tidlst[i];

  tret=pthread_create(&P_tid[i],&fifo_attr,s_thread_func,(void *)sptr);
  pthread_setaffinity_np(P_tid[i], sizeof(cpu_set_t), &cpuset);
  if (tret)
  {
    printf("ERROR. Return code from pthread_create() is %d\n", tret);
    exit(-1);
  }
  sptr++;
}
```

After all threads are created, main() program thread starts polling mouse for aperiodic events/termination signal through mouse() function.

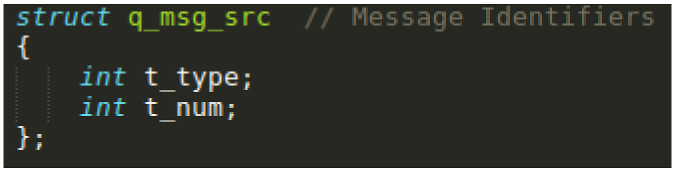


```
//=========================================
// Mouse polling
//=========================================
while(mouse_read!=3)
{
  mouse();
}
printf("out of main\n");
```

Threads :
========

All sender threads are calling common sender thread function.

Below object of below structure that is present in header file is created & passed for each thread execution & contains each thread’s specific identifier data (i.e. t_type & t_num)



```
struct q_msg_src  // Message Identifiers
{
    int t_type;
    int t_num;
};
```

Sender thread function runs in loop until termination signal is received through mouse. (i.e. mouse_read==3 for double-click of mouse)

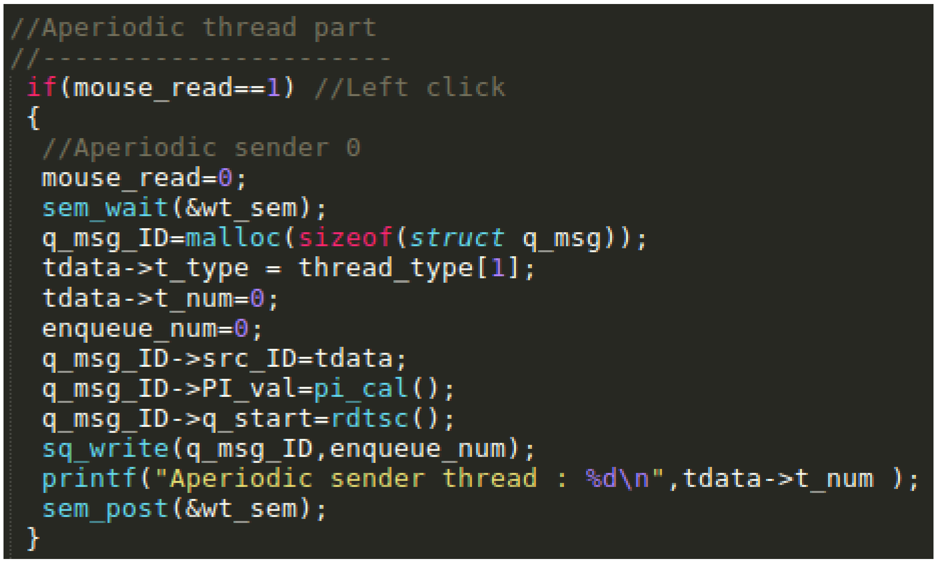
Every time any sender thread wakes & invokes sender functions, its ID is matched & respective sq_write call is made to write data in queue for that thread.

sem_wait() & sem_post is used around portion of code that writes to queue.

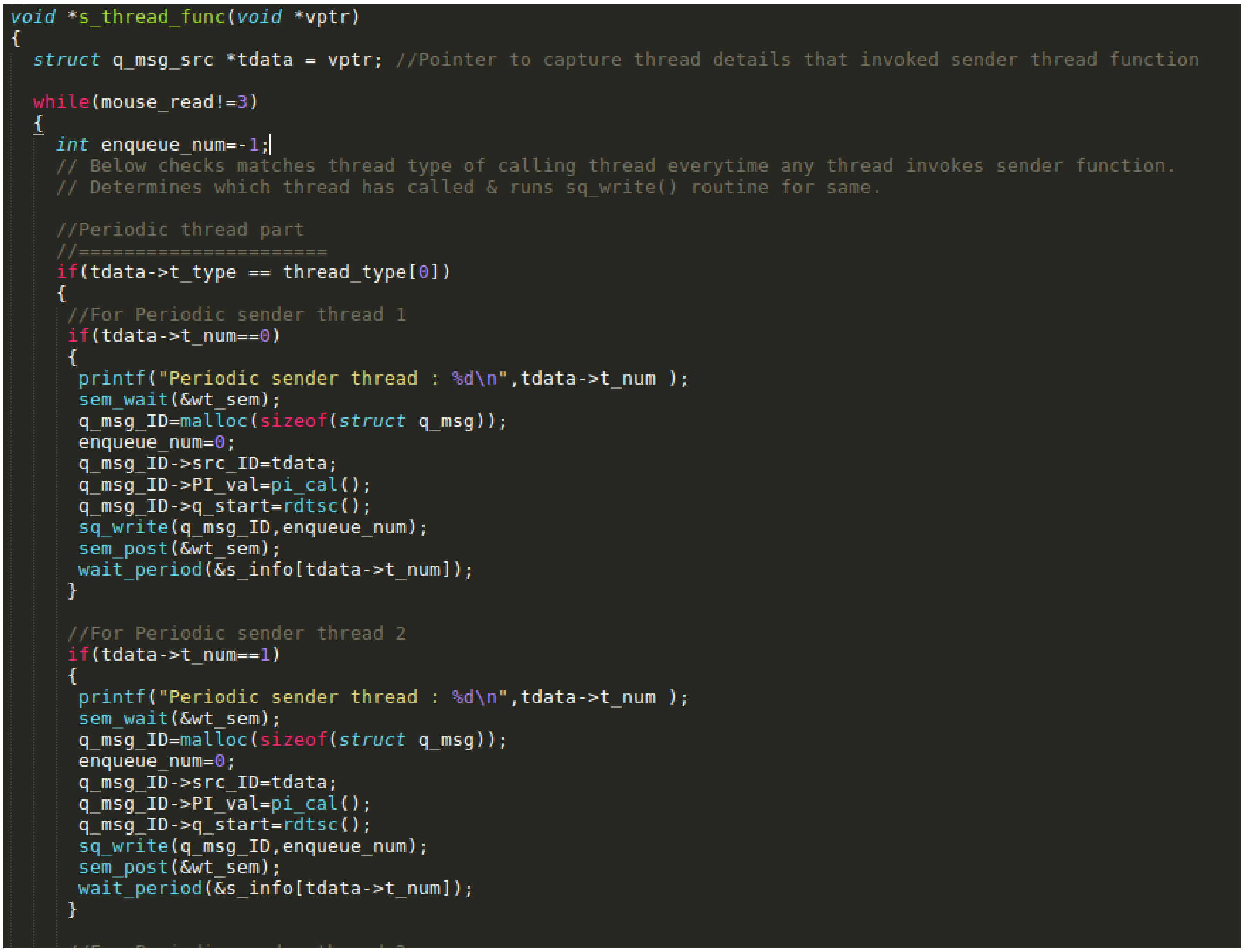
As message is pushed in queue, a start timestamp is assigned in q_start to it through RDTSC(). During dequeue, this value will be compared with rdtsc() recorded during dequeue which gives total queuing time.
wait_period() function is called which checks timerfd value to manage wake schedule as set during timer initializations.

Receiver thread executes with similar logic calling sq_read() function when it wakes.

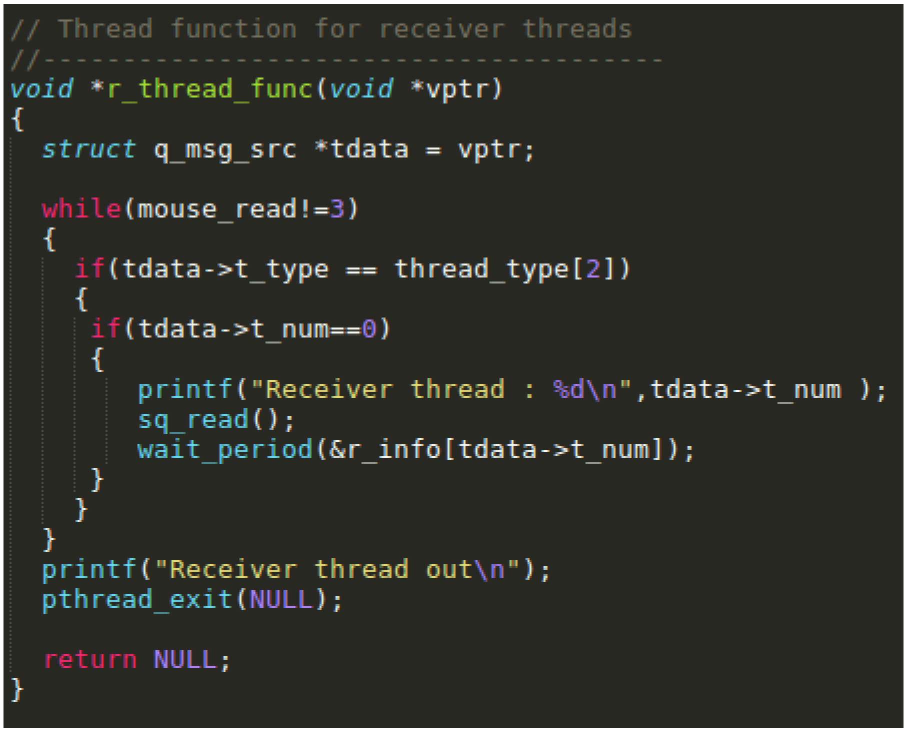
Aperiodic sender threads check for mouse left(mouse_read==1) or right click(mouse_read==2) & send out message to respective dataqueue.



```
//Aperiodic thread part
//----------------------
 if(mouse_read==1) //Left click
 {
  //Aperiodic sender 0
  mouse_read=0;
  sem_wait(&wt_sem);
  q_msg_ID=malloc(sizeof(struct q_msg));
  tdata->t_type = thread_type[1];
  tdata->t_num=0;
  enqueue_num=0;
  q_msg_ID->src_ID=tdata;
  q_msg_ID->PI_val=pi_cal();
  q_msg_ID->q_start=rdtsc();
  sq_write(q_msg_ID,enqueue_num);
  printf("Aperiodic sender thread : %d\n",tdata->t_num );
  sem_post(&wt_sem);
 }
```



```
void *s_thread_func(void *vptr)
{
  struct q_msg_src *tdata = vptr; //Pointer to capture thread details that invoked sender thread function

  while(mouse_read!=3)
  {
    int enqueue_num=-1;
    // Below checks matches thread type of calling thread everytime any thread invokes sender function.
    // Determines which thread has called & runs sq_write() routine for same.

    //Periodic thread part
    //======================
    if(tdata->t_type == thread_type[0])
    {
     //For Periodic sender thread 1
     if(tdata->t_num==0)
     {
      printf("Periodic sender thread : %d\n",tdata->t_num );
      sem_wait(&wt_sem);
      q_msg_ID=malloc(sizeof(struct q_msg));
      enqueue_num=0;
      q_msg_ID->src_ID=tdata;
      q_msg_ID->PI_val=pi_cal();
      q_msg_ID->q_start=rdtsc();
      sq_write(q_msg_ID,enqueue_num);
      sem_post(&wt_sem);
      wait_period(&s_info[tdata->t_num]);
     }

     //For Periodic sender thread 2
     if(tdata->t_num==1)
     {
      printf("Periodic sender thread : %d\n",tdata->t_num );
      sem_wait(&wt_sem);
      q_msg_ID=malloc(sizeof(struct q_msg));
      enqueue_num=0;
      q_msg_ID->src_ID=tdata;
      q_msg_ID->PI_val=pi_cal();
      q_msg_ID->q_start=rdtsc();
      sq_write(q_msg_ID,enqueue_num);
      sem_post(&wt_sem);
      wait_period(&s_info[tdata->t_num]);
     }
```



```
// Thread function for receiver threads
//--------------------------------------
void *r_thread_func(void *vptr)
{
  struct q_msg_src *tdata = vptr;

  while(mouse_read!=3)
  {
    if(tdata->t_type == thread_type[2])
    {
     if(tdata->t_num==0)
     {
        printf("Receiver thread : %d\n",tdata->t_num );
        sq_read();
        wait_period(&r_info[tdata->t_num]);
     }
    }
  }
  printf("Receiver thread out\n");
  pthread_exit(NULL);

  return NULL;
}
```

Header file :
=========

In this file there are four major functions i.e. sq_create(),sq_read(),sq_write() and sq_delete().

sq_create()

In this function an array of the required size is initialized which is used as circular dataqueue.

sq_write()

This is the function used by the periodic sender and aperiodic sender threads to write into the dataqueues. The function checks based upon the new_enqueue_number it identifies which dataqueue to write. And based on the flag whether the queue is full or not. A variable is used to keep the track of where to write , when this variable comes to the last position then upon incrementing it is agaun initialized to the intial postion.

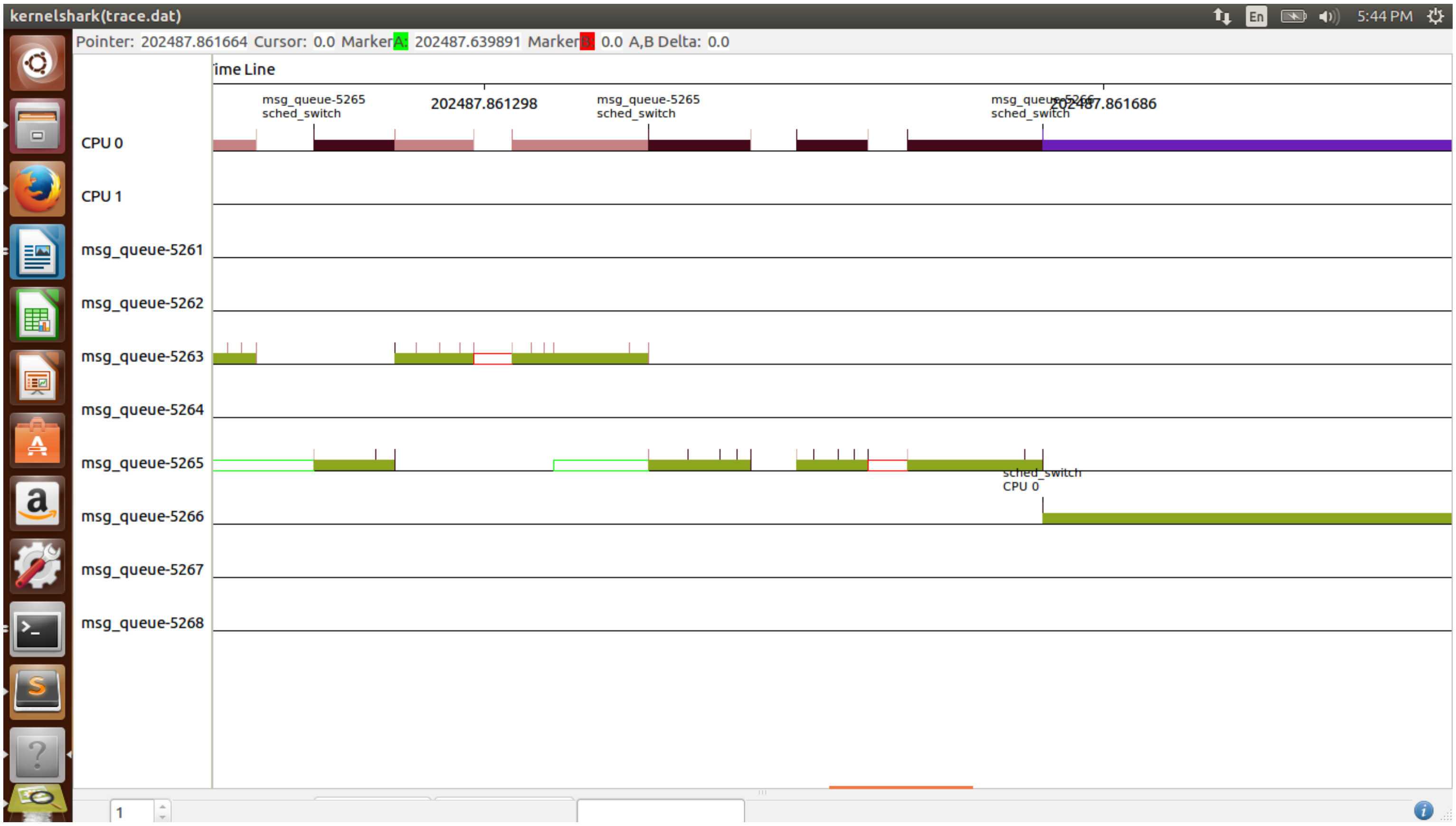
sq_read()

This function is used by the receiver thread to write into a singly linked list. In this function using flag and condition which check whether the dataqueue is empty or could be read. If could be read then the value is added to a linked list and the process is repeated until all the data of the queue are read.

sq_delete()

Upon calling this function it completes all the incomplete reads , calculates the number of messages sent and messages received and then calculates the averages of pi and queuing time and also the calculates the standard deviation of both pi and queuing time and frees any memory allocated by the malloc function.

Kernelshark report



As seen in above execution report from kernelshark, sched_switch is used as filter. Threads on running on single core 0.

PID 5261 is main.
PID 5262 is Periodic sender 1.
PID 5263 is Periodic sender 2.
PID 5264 is Periodic sender 3.
PID 5265 is Periodic sender 4.
PID 5266 is Aperiodic sender 1.
PID 5267 is Aperiodic sender 2.
PID 5268 is Receiver.

As seen in plot, 4 periodic threads have FIFO RT policy. When high priority thread of aperiodic sender 1 comes, it preempts the other running threads.