CHAPTER 5

CENTRAL QUEUE BASED EDF SCHEDULING AND INTERVAL TIMERS

5.1 INTRODUCTION

This chapter proposes security service enhancement, error detection, realtime scheduler (based on EDF algorithm). In this work, the proposed EDF scheduling algorithm have been analyzed and compared with rate monotonic algorithm to produce improved results upon various parameters in realtime environment. Queue based structure is implemented in both EDF realtime scheduler in packet data communication and realtime interval timer used in linux. Other timers like profile timer and virtual timers are also discussed.

5.2 CENTRAL QUEUE BASED EDF ALGORITHM

A priority based packet scheduling scheme is proposed due to its dynamic nature to the changing requirements, priority adjustment during run time, ability to schedule different types of data packets obtained by decomposing data sources (Video, image or audio sourcing) and ability to avoid improper allocation of data packets into queues. A packet is a unit of data that is transmitted across a packet-switched network. In wireless communication, enhancing the packet delivery may be implemented by using packet scheduling algorithms used to select which packet to be dropped or serviced that ensures packet delivery based on priority and fairness with minimum latency. The proposed central queue algorithm is efficient in terms of average packet waiting time, Packet delivery ratio and end –to-end delay (transmitting packets from source to destination with minimal delay).

Queuing algorithms allocate three independent quantities such as Bandwidth (Which packets gets transmitted), promptness (when packets transmitted) and buffer space (which and when packets get discarded by the scheduler). Giving more promptness (less delay) to the source users using less than their fair share of bandwidth is a good allocation strategy. The size of the packet and queue play a vital role in deciding the efficiency of queuing implementation. If queue filling rate is very high, it increases the length of queue and vice versa. Longer the size of packet,
larger the bandwidth shared by the source. When a packet arrives, it is buffered in an input queue according to its destination, priority class, and whether it has a single destination or multiple destinations. The data packets available at the ready queue have a deadline within which it should be delivered to the receiver. At the beginning of each fixed-length packet time (slot), the scheduler examines the contents of all input queues, the data packet which has the earliest deadline is sent first and packets leaving the input queue are buffered into the output queues where they await transmission to the external line.

The proposed algorithm control the order in which the packets are sent, the usage of buffer space, the rate at which the source sends packets, determine the way in which packets from two different sources interact with each other. Realtime packets are processed in the way as First Come First served and a FIFO strategy can be used to evict packets before they expire. Figure 5.1 shows the model of data packet networking arriving into the central queue implemented in this work. Here, $src_{1}^{(1)}$, $src_{2}^{(1)}$ are the data packets from source 1 and source 2 respectively arriving into the queue in the FIFO manner.

![Figure 5.1 Queue model](image)

Figure 5.1 Queue model

![Figure 5.2 State diagram](image)

Figure 5.2 State diagram
Figure 5.2 shows the state diagram of central queue model. Task 1 is the queue filling rate and data packets are arriving into the queue, task 2 is the servicing rate of the queue. Whenever the queue is full, it is indicated by the event flag and task 1 will be in the delayed state. Now data packets are ready to be serviced, task 2 is processed and whenever the queue is released data packets are arriving, i.e., task 1 is running. The EDF scheduling unit uses the above central queue model in which the queue will be searched for the new task closest to its deadline whenever a task is finished or new task is released.

### 5.2.1 BLOCK DIAGRAM IMPLEMENTATION

Figure 5.3 shows the set up implemented using security protocol (Blowfish) and error detection coding schemes. This realtime application has been developed and run on cortex M3 LPC 1788 processor using µc/os-II.

![Block Diagram](image)

**Figure 5.3 Security and error detection concepts**

i) **ENCRYPTION LAYER**

Module I employs blowfish encryption algorithm, a symmetric block cipher that can be effectively used for encryption and safeguarding of data and using 64 bits of data blocks and a variable size key maximum up to 448 bits. It comprises of Feistel Network having 16 times iterative operations of a simple encryption function. The prime characteristics of Blowfish
algorithm is that it includes key dependent S-boxes and has a complex key schedule which makes the algorithm stronger. Blowfish uses a large number of sub keys. These keys must be pre computed before any data encryption or decryption. Data packet size of 256 bytes is shown as an example which is of 1024 bytes in encrypted form.

1. The P-array consists of 18 32-bit sub keys:
P1, P2... P18.

2. There are four 32-bit S-boxes with 256 entries each:
S1,0, S1,1,..., S1,255;
S2,0, S2,1,..., S2,255;
S3,0, S3,1,..., S3,255;
S4,0, S4,1,..., S4,255.

The data block of 64 bits are first divided into two halves of 32 bits each. Each line in the diagram of the Blowfish algorithm represents 32 bit data. It uses two sub key arrays 18-entry P-array and 256-entry S-boxes. The S-boxes convert the 8 bit input into 32 bits output. One entry of P-array is compulsory for each of 16 rounds as shown in the Figure 5.4. The remaining two P-array entries are used after the final round to separately XOR the outputs of each of the halves of the data block of 32 bits. In the function F, four S-boxes and two types of bit operations: XOR and addition of modulo $2^{32}$ are used. The function F first divides the input of 32 bits into four S-boxes of consisting 8 bits each. The outputs of first and second S-boxes are first added to modulo $2^{32}$ and the output is XOR ed with the third S-box output. The result of XOR operation and the output of fourth S-box are finally added to modulo $2^{32}$ and we get the final 32 bit output from the function F. The round function operation is shown in Figure 5.5.
Figure 5.4 Blowfish Algorithm

Figure 5.5 Round function in blowfish
ii) SCHEDULER UNIT

The encrypted and secured data will be of 1024 bytes and is processed in module II for assigning priority number which is a realtime central queue based EDF scheduler of µc/os-II. Packet scheduling is a process defined as decision making to select or drop the packet. Scheduler is used to schedule the packets and having hard time to handle when all packets coming in with high packet rate, when bandwidth is too low and packet size is large. The scheduler will make decision to select the packets based on EDF algorithm. Dropping of packet depends on packet size, bandwidth, packet arrival rate and deadline of packet. It is by default that not all packets may reach the destination and some of the packets may be dropped along the way with respect to the above network characteristics.

iii) ERROR DETECTION MODULE

Packets may be lost or corrupted in transmission on the wireless network. The error detecting module receiving a corrupted packet can detect the error and discard the packet. Cyclic Redundancy Check (CRC) implemented for error detection shown in module III commonly used in digital networks and storage devices to detect accidental changes to raw data. It is based on the theory of cyclic codes. The use of systematic cyclic codes, which encode messages by adding a fixed-length check value is for the purpose of error detection in communication networks. On retrieval, the calculation is repeated and corrective action can be taken against presumed data corruption if the check values do not match. In this way, all the data packets are manipulated using the above implementation.

By implementing security and error detection schemes, the data structure for packet 1 of source 1 is shown in Figure 5.6 as an example.

![Data structure of packet of source 1](image)
5.3 QUEUING IMPLEMENTATION

5.3.1 DECOMPOSING DATA INTO STREAM OF PACKETS

In packet data transmission, sending of data packets by an individual source may be delayed and it automatically decreases fairness of bandwidth of individual source. When another source sends packets too quickly, it merely increases the length of the queue. This algorithm makes a tradeoff between these two streams of data packets by combining two sources of dedicated network. Figure 5.7a to 5.7f show how user data is decomposed into stream of packets by undergoing several steps from non-realtime to realtime implementation.

Figure 5.7a Separate channels for src 1 and src2

Figure 5.7b Common channel
Figure 5.7c Error detection added to data structure

Figure 5.7d Security and error detection

Figure 5.7e Packet id assignment

Figure 5.7f Realtime multiple source data structure
5.3.2 PRIORITY ASSIGNMENT

Table 5.1 and 5.2 provide the information of allocating priority for source 1, source 2 and its corresponding data packets. Realtime data packets are given highest priority among all the data packets in the ready queue, so higher priority packets are processed first and then it has to be delivered to the receiving side with minimized end-to-end delay.

**Table 5.1 Starting Priority for Multiple Source Packets**

<table>
<thead>
<tr>
<th>Sources</th>
<th>Priority Allocation (Initial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>65535</td>
</tr>
<tr>
<td>Src2</td>
<td>64511</td>
</tr>
</tbody>
</table>

The user associated with a packet could refer to the source of a packet, its priority allocated by referring to its ID from the table 5.2 given below.

**Table 5.2 Priority of Packets for Source 1 and Source 2**

<table>
<thead>
<tr>
<th>Src1 Packet ID</th>
<th>Priority Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65535</td>
</tr>
<tr>
<td>2</td>
<td>65534</td>
</tr>
<tr>
<td>3</td>
<td>65533</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src2 Packet ID</th>
<th>Priority Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>64511</td>
</tr>
</tbody>
</table>
5.3.3 SCHEDULER IMPLEMENTATION

Figure 5.8 implements EDF scheduling based central queue algorithm under realtime environment of μc/os-II.

The servicing and dropping of the packets will be based on several parameters such as bandwidth, packet arrival rate, and packet deadline and packet size. Scheduling of packets will be done in a scheduler and the scheduler will find it difficult to handle each and every packet due to the high packet rate, low bandwidth and less packet size and the scheduler will select certain packets based on central queue based algorithm. Table 5.3 and Table 5.4 explain how the scheduler implements selection of data packets for a particular time period.

Table 5.3  Arrival of Source 1 and Source 2 Packets into Queue
<table>
<thead>
<tr>
<th>Queue Element</th>
<th>Packet Number</th>
<th>Queue Buffer ID</th>
<th>Priority Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src2</td>
<td>1</td>
<td>1</td>
<td>64511</td>
</tr>
<tr>
<td>Src1</td>
<td>1</td>
<td>1</td>
<td>65535</td>
</tr>
<tr>
<td>Src1</td>
<td>2</td>
<td>1</td>
<td>65534</td>
</tr>
<tr>
<td>Src2</td>
<td>2</td>
<td>1</td>
<td>64510</td>
</tr>
<tr>
<td>Src1</td>
<td>3</td>
<td>2</td>
<td>65533</td>
</tr>
<tr>
<td>Src1</td>
<td>4</td>
<td>3</td>
<td>65532</td>
</tr>
</tbody>
</table>

Table 5.4  Queue State, Packet Selected vs Time

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>[Multiple Packets ]</th>
<th>Selected Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>src_2^{(1)}</td>
<td>src_2^{(1)}</td>
</tr>
<tr>
<td>24</td>
<td>$\text{src}_2^{(2)}$</td>
<td>$\text{src}_1^{(3)}$</td>
</tr>
<tr>
<td>----</td>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td></td>
<td>$\text{src}_1^{(3)}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{src}_1^{(4)}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32</th>
<th>$\text{src}_2^{(2)}$</th>
<th>$\text{src}_1^{(4)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{src}_1^{(4)}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{src}_1^{(5)}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>40</th>
<th>$\text{src}_2^{(2)}$</th>
<th>$\text{src}_1^{(5)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{src}_1^{(5)}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{src}_1^{(3)}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{src}_1^{(6)}$</td>
<td></td>
</tr>
</tbody>
</table>
5.4 REALTIME COMPARISON OF RM AND EDF

In this work, Rate Monotonic (RM) and Earliest Deadline First (EDF) priority scheduling algorithms are implemented using an example. Both these algorithms are priority based scheduling algorithms for periodic tasks with hard deadlines. Table 5.5 shows the example task set. The following assumptions are taken in this implementation of the algorithms.

Arrival times for first instances of all tasks are assumed to be 0.

For each job $k$, deadline $= \text{arrival time} + \text{period}$. All instances of a periodic task should have the same computation time.

All instances of a periodic task should have the same relative deadline, which is equal to the period.

**Table 5.5 Example task set**

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival time(for first instance)</th>
<th>Period (ms)</th>
<th>Execution time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>0</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>0</td>
<td>20</td>
<td>7</td>
</tr>
</tbody>
</table>

The job list derived from Table 5.5 under the assumption deadline $= \text{arrival time} + \text{period}$ for any job $k$ is shown in table 5.6.

**Table 5.6 Job list for table 5.5**

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival time (a) in ms</th>
<th>Execution time (ms)</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_1(\tau_1)$</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$k_2(\tau_2)$</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>$k_3(\tau_1)$</td>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>$k_4(\tau_2)$</td>
<td>5</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>$k_5(\tau_1)$</td>
<td>8</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>$k_6(\tau_2)$</td>
<td>10</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>$k_7(\tau_1)$</td>
<td>12</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>$k_8(\tau_3)$</td>
<td>0</td>
<td>7</td>
<td>20</td>
</tr>
</tbody>
</table>
5.4.1 GRAPHICAL IMPLEMENTATION

EDF (Jobs)

<table>
<thead>
<tr>
<th>k_9(\tau_2)</th>
<th>15</th>
<th>2</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>k_{10}(\tau_1)</td>
<td>16</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 5.9 Timeline execution of EDF algorithm

RM (Jobs)

Figure 5.10 Timeline execution of RM algorithm

Figure 5.9 & 5.10 show graphical implementation of all jobs schedulable under EDF and RM listed in table 5.6.
5.4.2 CALCULATION OF RM ALGORITHM PARAMETERS

i) WORST CASE RESPONSE TIME

For the given task set $\tau_1=4\text{ms}$, $\tau_2=5\text{ms}$, $\tau_3=20\text{ms}$, $C_1=1\text{ms}$, $C_2=2\text{ms}$ and $C_3=7\text{ms}$, the worst case response time of RM algorithm can be computed using the equation (5.1),

$$R_{i}^{n+1} = C_i + \sum_{j \in hp(i)} \left[ \frac{R_i}{T_j} \right]^n C_j \quad \text{(5.1)}$$

Here, $C_i$ is the maximal execution time, $R_i$ is the response time for task $\tau_i$ and the set $hp(i)$ consist of all tasks with higher priority than task $\tau_i$ and $T_j$ is the time period of the highest priority task.

Priorities are assumed to be unique and tasks will not voluntarily suspend themselves. The terms in the summation express how many times each higher priority task will interfere with task $\tau_i$ multiplied with the higher priority task’s execution time, $C_j$. To solve the above equation the response time, $R_i$ must be factored out.

When priorities are set as fixed, the response times of tasks can be calculated. It is begun with the highest priority task $\tau_1$. No other task will interfere its execution, so the response time will be equal to its maximal execution time 1.

$R_{\tau_1} = 1$ ; (Best case response time of first task) the response time for the next highest priority task $\tau_2$ is then calculated. Only one task $\tau_1$, has higher priority and will interfere with $\tau_2$. Response time for $\tau_2$ is calculated as:

$R_{\tau_2}^1 = 2 + 2/4*1 = 2 + 1 = 3$

$R_{\tau_2}^2 = 2 + 3/4*1 = 2 + 1 = 3$

(First iteration response)

$R_{\tau_2}^1 = R_{\tau_2}^2 = 3$

The sequence of iterations terminates with the response time 3 for task $\tau_2$. 

15
Next task to consider is $\tau_3$, which will be interfered by two higher priority tasks. The response time iterations are:

\[ R_{\tau_3}^1 = 7 + \frac{7}{4} \times 1 + \frac{7}{5} \times 2 = 7 + 2 + 3 = 12 \]
\[ R_{\tau_3}^2 = 7 + \frac{12}{4} \times 1 + \frac{12}{5} \times 2 = 7 + 3 + 6 = 16 \]
\[ R_{\tau_3}^3 = 7 + \frac{16}{4} \times 1 + \frac{16}{5} \times 2 = 7 + 4 + 8 = 19 \]
\[ R_{\tau_3}^4 = 7 + \frac{20}{4} \times 1 + \frac{20}{5} \times 2 = 7 + 5 + 8 = 20 \]
\[ R_{\tau_3}^3 = R_{\tau_3}^4 = 20 \]

ii) **BEST CASE RESPONSE TIME**

The best case response time of RM algorithm can be computed using the equation (5.2) as shown.

\[ R_{i+1}^b = C_i + \sum_{j \in \text{hp}(i)} \left[ \frac{R_i}{T_j} - 1 \right] C_j \quad \text{(5.2)} \]

It is calculated for all the jobs using the same iterative method used for worst case response time.

\[ R_{\tau_1}^b = 1; \]
\[ R_{\tau_2}^b = 2; \]
\[ R_{\tau_3}^b = 12; \]

iii) **OUTPUT JITTER**

Output jitter may be computed for each task if response-time analysis is done. Let $R_i$ and $R_i^b$ denote, respectively, the worst-case and best-case response times of task $\tau_i$. The jitter, $J_i$ is then given by,

\[ J_i = R_i - R_i^b \quad \text{(5.3)} \]

$J_{\tau_1} = 0; J_{\tau_2} = 1$ and $J_{\tau_3} = 8$ respectively under RM scheduling for the given task set.
iv) LATENCY

Another parameter that it is important in control applications is the input-output latency. Assuming that a control task $\tau_i$ acquires inputs at the beginning of each instance and delivers control outputs at the end, the maximum input-output latency is defined as,

$$L_i = \max_i (\text{finish-time} - \text{start-time}) \quad \text{--------------- (5.4)}$$

It is also defined as,

$$L_i = R_i^b \quad \text{--------------- (5.5)}$$

Under RM scheduling, 

$L_{\tau_1}$=1; $L_{\tau_2}$=2 and $L_{\tau_3}$=12 respectively.

5.4.3 CALCULATION OF EDF ALGORITHM PARAMETERS

i) WORST CASE RESPONSE TIME

Under EDF scheduling, the calculation of worst case response-time analysis is more complicated and based on the theorem “the worst case response time of task set $\tau_i$ is found in a busy period in which all tasks are released synchronously at the beginning of the period at the maximum rate”.

To calculate worst-case response time of task $\tau_i$ under EDF scheduling, consider equations given below:

$$R_i = \max[C_i, \max \{ Li(a) - a \}] \quad \text{----------------------------- (5.6)}$$

$$a \geq 0$$

Here’ a’ denotes the arrival time or release time of task and $d=a+D_i$, $Li$ is the constant delay of arrival time, $D_i$ is the relative deadline of the task, $d$ is the absolute deadline and $R_i$, the response time of task.

Where the busy interval $L_i (a)$ is given by the equation,

$$L_i (a) = W_i[a, Li(a)] + \left(1 + \left[ \frac{a}{T_i} \right]\right) Ci \quad \text{----------------------------- (5.7)}$$
The period $T_i$ of the periodic task $\tau_i$ is a fixed time interval between release times of consecutive jobs and the higher-priority workload $W_i \{a, L_i (a)\}$ is given by,

$$W_i[a,L_i(a)] = \sum_{j \in L_i[a+Di]} \left[ \min \left( \frac{L_i(a)}{T_j}, 1 + \frac{a + Di - Dj}{T_j} \right) \right] C_j$$

$$---------- (5.8)$$

For task $\tau_1$, $W_i \{a, L_i (a)\} = 1; L_i (a) = 1$ and $R_{\tau_1} = 1$.

Similarly, $R_{\tau_2} = 2.75$ and $R_{\tau_3} = 13$ respectively.

**ii) BEST CASE RESPONSE TIME**

The best case response time of EDF algorithm can be computed using the equation as shown.

$$R_{i+1}^n = C_i + \sum_{j \in jhp(i)} \left[ \frac{R_i^n}{T_j} - 1 \right] C_j$$

$$---------- (5.9)$$

But for tasks with short period, response time will be equal to computation time as per equation,

$$R_i^b = C_i$$

$$---------- (5.10)$$

So, $R_{\tau_1}^b = C_1 = 1$;

$R_{\tau_2}^b = 1.25$;

$R_{\tau_3}^b = 7 + \min \left[ \frac{7.20 - 5}{5} - 1 \right] + \min \left[ \frac{7.20 - 4}{4} - 1 \right] = 8.55$;

**iii) OUTPUT JITTER**

The jitter calculation of EDF algorithm can be computed using the same equation as shown.

$$J_i = R_i - R_i^b$$

$$---------- (5.11)$$

$J_{\tau_1} = 0, J_{\tau_2} = 1.5$ and $J_{\tau_3} = 4.5$ respectively for the given task set.

**iv) LATENCY**

Under EDF scheduling,

$L_{\tau_1} = 1; J_{\tau_2} = 1.25$ and $L_{\tau_3} = 8.55$ respectively.
5.5 INTERVAL TIMERS FOR PROCESS HANDLING

Signals, or software interrupts, are external, asynchronous events used to alter the course of a program. These may occur at any time during the execution of a program. Because of this, they differ from other methods of inter-process communication. The alarm () function may cause the system to generate a SIGALRM signal for the process after the number of realtime seconds specified by seconds have elapsed. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

If seconds is 0, a pending alarm request, if any, is canceled.

Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner. If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time at which the SIGALRM signal is generated.

In present-day Linux, each process is equipped with three timers, and timers are identified by a value of type interval_timer:

**ITIMER_REAL** Real time (sigalrm).

**ITIMER_VIRTUAL** User time (sigvtalrm).

**ITIMER_PROF** User time and system time (sigprof).

The state of a timer is described by the interval_timer_status type which is a record with two fields (each a float) representing time:

- The field it_interval is the period of the timer.
- The field it_value is the current value of the timer; when it turns 0, the signal sigvtalrm is sent and the timer is reset to the value in it_interval.

A timer is therefore inactive when its two fields are 0. Table 5.7 shows state descriptions of interval timers.

**TABLE 5.7 TIMER STATE DESCRIPTIONS**

<table>
<thead>
<tr>
<th>it_value</th>
<th>it_interval</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>≠0</td>
<td>≠0</td>
<td>Indicates time to the next timer expiration and reloading it_value</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>Disables the timer</td>
</tr>
</tbody>
</table>
5.5.1 PROCESS SPECIFIC INTERVAL TIMERS

Linux supports process specific interval timers. The system provides each process with three interval timers, each decrementing in a distinct time domain. When a timer expires, a signal is sent to the process, and the timer is reset to the specified interval (if nonzero). A process can use these timers to send itself various signals each time that they expire. Three sorts of interval timers are supported.

i) \textbf{ITIMER\_REAL}

This timer ticks only in realtime and when it expired, the process is sent SIGALRM signal.

ii) \textbf{ITIMER\_VIRTUAL}

This timer ticks only when the process is executing and delivers SIGVTALRM upon expiration.

iii) \textbf{ITIMER\_PROF}

It decrements both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER\_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

One or all interval timers may be running and Linux keeps all of the necessary information in the process’s task\_struct data structure. System calls can be made to set up these interval timers and to start them, stop them, and read their current values. The virtual and profile timers are handled the same way. In every clock tick, the current process’s interval timers are decremented and, if they have expired, the appropriate signal is sent.
5.5.2 QUEUING IMPLEMENTATION FOR REALTIME INTERVAL TIMERS

Realtime interval timers are a little different and use the timer mechanism. Each process has its own timer_list data structure and when the real interval timer is running, this is queued on the system list. When the timer expires, the timer half handler removes it from the queue and calls the interval timer handler. This generates the SIGALARM alarm signal and restarts the interval timer, adding it back into the system timer queue. A task queue is a list of tasks, each task being represented by a function pointer and an argument. When a task is run, it receives a single void * argument and returns void. The pointer argument can be used to pass along a data structure to the routine, or it can be ignored. The queue itself is a list of structures (the tasks) that are owned by the kernel module declaring and queuing them. The module is completely responsible for allocating, deallocating the structures and static structures are commonly used for this purpose.

```c
struct tq_struct {
    struct tq_struct *next; /* linked list of active bh's */
    int sync;               /* must be initialized to zero */
    void (*routine)(void *); /* function to call */
    void *data;             /* argument to function */
};
```

The "bh" is meant for bottom half, i.e, "half of an interrupt handler". A bottom half is a mechanism provided by a device driver to handle asynchronous tasks which, usually, are too large to be done while handling a hardware interrupt. The most important fields in the data structure shown above are routine and data. To queue a task for later execution, it is needed to set these fields before queuing the structure, while next and sync should be cleared. The sync flag in the structure is used by the kernel to prevent queuing the same task more than once, because this would corrupt the next pointer. Once the task has been queued, the structure is considered "owned" by the kernel and should not be modified until the task is run.
i) OPERATIONS PERFORMED ON TASK QUEUES AND STRUCTTQ_STRUCTS

The other data structure involved in task queues is task_queue, which is currently just a pointer to structtq_struct. Task_queue pointers should be initialized to NULL before use. The following list summarizes the operations that can be performed on task queues and structtq_structs.

**declare_task_queue (name);**

This macro declares a task queue with the given name, and initializes it to the empty state.

**Intqueue_task (structtq_struct *task, task_queue *list);**

As its name suggests, this function queues a task. The return value is 0 if the task was already present on the given queue, nonzero otherwise.

**voidrun_task_queue (task_queue *list);**

This function is used to consume a queue of accumulated tasks. It is not needed to call it unless it is declared and maintained own queue.

i) STRUCTURAL DEFINITION OF TIMER VALUES

Timer values are defined by the following structures:

```c
struct timerval {
    struct timerval it_interval; /* Interval for periodic timer and period value, 0 for one-shot timer */
    struct timerval it_value;    /* Time until next expiration and initial value, 0 to disable timer */
};

struct timerval {
    time_t tv_sec;    /* seconds */
    suseconds_t tv_usec; /* microseconds */
};
```
The function getitimer() fills the structure pointed to by curr_value with the current value (i.e., the amount of time remaining until the next expiration) of the timer specified. The subfields of the field it_value are set to the amount of time remaining on the timer, or zero if the timer is disabled. The it_interval field is set to the timer interval (period); a value of zero returned in (both subfields of) this field indicates that this is a single-shot timer. The function setitimer() sets the specified timer to the value in new_value. If old_value is non-NULL, the old value of the timer (i.e., the same information as returned by getitimer()) is stored there. Timers decrement from it_value to zero, generate a signal, and reset to it_interval. A timer which is set to zero (it_value is zero or the timer expires and it_interval is zero) stops. Both tv_sec and tv_usec are significant in determining the duration of a timer.

Timers will never expire before the requested time, but may expire some (short) time afterward, which depends on the system timer resolution, and on the system load. Upon expiration, a signal will be generated and the timer is reset. If the timer expires while the process is active (always true for ITIMER_VIRTUAL), the signal will be delivered immediately when generated. Otherwise, the delivery will be offset by a small time dependent on the system loading.

5.6 DEMONSTRATION OF DIFFERENT ALARM SIGNALS

Here the real timer measures the wall clock time and thus ticks whether the associated process is running or not. Virtual timer measures the time, the process is running in user mode and thus ticks while the process is running, but not during the execution of system calls by the kernel on behalf of the process. The profile timer measures the time, a process is running in user mode and the time that system calls take which are being executed on behalf of the process (kernel mode). Given these three timers, it is possible to compute realtime, cpu time (time in user and kernel mode), user time and kernel time. The kernel time is computed by subtracting virtual time from the profile time for a process.

In order to demonstrate the different alarm signals, Fibonacci series of a given number is found in child processes for calculating real, profile and virtual times using alarm signals. The real, CPU, and kernel times are calculated using the following formulae:

- Realtime = time elapsed from real timer
- CPU time = time from profile timer
- User time=time elapsed from virtual timer
- Kernel time=time elapsed from (profile timer-virtual timer)

The flow of execution of the main function is,
- Get the number to find Fibonacci series. By default the number is 30.
- Initialize the ITIMER for real, virtual and profile timers as follows,

```c
P_realt.it_interval.tv_sec=MAXSEC
P_realt.it_interval.tv_usec=MAXUSEC
P_realt.it_value.tv_sec=MAXSEC
P_realt.it_value.tv_usec=MAXUSEC
P_virtt.it_interval.tv_sec=MAXSEC
P_virtt.it_interval.tv_usec=MAXUSEC
P_virtt.it_value.tv_sec=MAXSEC
P_virtt.it_value.tv_usec=MAXUSEC
P_proft.it_interval.tv_sec=MAXSEC
P_proft.it_interval.tv_usec=MAXUSEC
P_proft.it_value.tv_sec=MAXSEC
P_proft.it_value.tv_usec=MAXUSEC
```

As an example, the function, tv_usec needs to be incremented once in every microseconds. The system can track the passage of time by counting the number of interrupts that have occurred since system boot.
The signal function provides a simple interface for establishing an action for a particular signal. The function and associated macros are declared in the header file `signal.h'.

Data Type: sighandler_tsignal (int sig, sighandler_t handler)

Set the signal handler for alarm signals as follows,

```
Signal (SIGALRM, p_realt_handler);
Signal (SIGVTALARM, p_virtt_handler);
Signal (SIGPROF, p_proft_handler);
```

Signal handlers take one integer argument specifying the signal number, and have return type void. So, handler functions are defined as follows:

```c
void(*signalhandler_t (int)

The signal handler format is as follows,

```
Void p_realt_handler ()
{
  P_realt_secs++;
  /*
   * Printf ("p_realt_handler\n");
   * fflush (stdout);
  */
  Signal (SIGALRM, p_realt_handler;

The itimers use kernel space to keep track of
itimer real: passage of realtime
itimer virtual: passage of virtual time (only when the process is running)
itimer profile: Time of the process is actually running and the time the kernel is doing work on the process’s behalf.

All three timers work by counting down from a value set with the settimer system call. When zero is reached, all three will generate a specific signal (and may reset when reset time is specified). It should also be noted that a child process does not inherit its parent’s timers.

Initialize and Set the different ITIMERs as follows,

```c
if (settimer(ITIMER_VIRTUAL,&p_virtt,(structitimerval*)0)==-1)  
Perror (“parent virtual timer”);
if (settimer(ITIMER_REAL,&p_realt,(structitimerval*)0)==-1)  
Perror (“parent virtual timer”);
if (settimer(ITIMER_PROF,&p_proft,(structitimerval*)0)==-1)  
Perror (“parent virtual timer”);
```

Creating a process is one of the responsibilities of the operating system kernel. Parent process is the process that creates another process and the created process by parent process is named as the child process. Child process inherits most of parent process’s attributes, such as file descriptors. Each process may create more than one child processes. But child process may only has single parent process. Process does not have a parent, when it was created directly by the kernel. When parent process terminated, its child process will be leaving as an orphan process. But after child process become orphan it will be adopted by the kernel process.

A system call, or system service, is usually provided so that it can be used by programmers and/or within the operating system to create a new process. The name and arguments of this system call varies from one operating system to another. For the Linux operating system, the fork() system call creates new processes. The syntax for the fork system call is:

```c
retval = fork();
```
When the fork() system call is called, a child process is created for the calling process. The process within which the fork() system call is called is actually the parent process of the process being created. The child process is a copy of the parent process, except for some attributes. The returned value retval is the integer zero for the child process and it is the child’s process ID for the parent process in order that the parent knows child’s process ID. The parent and child processes will continue to execute the same code after the fork () instruction. However,

it is possible to check the retval value and let the parent and child processes follow different paths of instructions. It is safe to assume that an exact copy of the parent process’s data and code (program instructions), is made for the child process.

Create two child processes using fork () system call,

Pid1=fork ();

Process identification information, is recorded right after the process is created or born. Usually, every process is also given a unique numeric identifier, called a process ID, in order to make future references to it easier and unambiguous.

A parent process may wait for a child process to complete its task using wait () or wait pid () system calls, but a child process cannot wait for its parent to complete a task. This means that a parent process is able to create a child process to do a certain task and to report the status of its completion to the parent. However, this is not possible the other way around.

The code for a kernel timer that reports realtime, cpu time, user time and kernel time on the operation of fibanocci method is shown below. It is used to keep track of the parent and two child times using fork ()

Check for parent process, find the Fibonacci series and wait for the child completion status. Get the ITIMERs values and display the elapsed times of different alarm signals.
else {/* this is the parent*/

    // Start parent on the fibonacci program
    fib = Fibonacci (fibarg);

    // Wait for children to terminate
    waitpid (0, &status, 0);
    waitpid (0, &status, 0);

    // Read parent itimer values and report them
    gettimer (ITIMER _PROF, &p_proft);
    gettimer (ITIMER _REAL, &p_proft);
    gettimer (ITIMER _VIRTUAL, &p_proft);
    printf ("\n");
    printf("parent fib=%ld, realtime=%ld sec,%ld millisec\n", fib, p_realt_secs,
           elapsed_usecs(p_realt.it_value.tv_sec,
                         p_realt.it_value.tv_usec)/1000;
    printf ("parent fib=%ld, cpu time=%ld sec,%ld millisec\n", fib, p_proft_secs,
             elapsed_usecs(p_proft.it_value.tv_sec,
                           p_proft.it_value.tv_usec)/1000;
    printf ("parent fib=%ld, user time=%ld sec,%ld millisec\n", fib, p_virtt_secs,
             elapsed_usecs(p_virtt.it_value.tv_sec,
                           p_virtt.it_value.tv_usec)/1000;
    printf ("parent fib=%ld, kernel time=%ld sec, %ld millisec\n", fib, p_kernel_secs,
fflush () is used to release the current contents of the output buffer to the system for whatever processing the system has for it. This is useful if something is waiting for the output.

5.7 SUMMARY

This chapter presents central queue based EDF scheduling scheme for secured data transmission under realtime environment to increase throughput and providing optimal operation. This queue based implementation is done in interval timers in linux, two child processes are created using fork () system call, Checked for parent process, the Fibonacci series was found , the ITIMERS values and the elapsed times of different alarm signals are displayed.

The study of various scheduling algorithms along with the comparison between different realtime scheduling algorithms has been done in the thesis. Proposed preemptive EDF algorithm is compared with RM algorithm under various performance metrics. Both RM and EDF are very well suited to work in low load conditions and achieving control of response time, response time jitter, best case and worst case response times.