PIKACHU

A REAL-TIME OPERATING SYSTEM

By
Saugata Das
Gaurav Vashishta
PESIT, Bangalore-560085

1. ABSTRACT

The water supply of a city is an essential service for its citizens. Similarly the fuel and gas supply of cities and countries forms the backbone of their economies. On numerous occasions we hear that an oil pipeline has burst in a remote area, and it effects the entire country as fuel supply lines get disrupted. The losses to the company maintaining the pipeline are major but the over all effect on the economy if even more severe.

Imagine the water and sewage pipeline network of a city. A small blockage in a major pipe leaves entire areas of the city without water. Let us then ask how the flow actually occurs. There must be a source and thereby a destination of the water flows. Such a dense network would have a number of routes from one source to its destination. Therefore the decision to be made in such a situation is about the route to be selected.

What happens when a leakage occurs in the pipelines? At present the only solution is to manually check the junctions and find the source of leakage. This itself is a very time consuming process and may even take days. Once the source of leakage is found, information about it has to be transmitted to a control center, which then stops the flow in the pipe and redirects it manually. This process involves a lot of wastage of resource and also causes unbearable discomfort to the users of the service. Mapping this problem to a larger network, like the gas lines of major cities and the fuel pipeline system of a country would bring out the immense nature of the above problem. One might ask what could be the solution to this problem.

The major goals for any solution would be to reduce the resource wastage and minimize the response time of reaction to any fault occurring in such system. One possibility is to automate the entire monitoring and response system. The technology exists to carry out such a task. At present similar solutions exist for computer networks. What is needed is an application, which utilizes the latest technology and years of experience in managing networks and integrates it with the various networks to provide the needed features.

As the major goals have been identified as minimization of resource wastage and response time, we need to have a system, which responds in real time and is self-healing. Meaning that any disruption in the system be automatically detected and corrected within a bound time period.

The real-time solution that we provide essentially implements all the goals specified above. According to our solution a real time operating system is to be designed with the specific purpose of running the system and reacting to any fault within specific time period.

In this system every pipeline will have a monitoring system which will tell a control center whether the line is functioning properly or not. The administrator will be able to view the map of all monitors specifying all line in his purview. Not only this he will also be able to view the delay involved in transportation of fluid using each line separately. Thereby, a whole map of the system exists in front of him.

Let us say that the specific system is that of an oil pipeline. There are three pipelines, which can transport oil from a source to a destination. One of these lines is being used on grounds that it has least delay. Now this line springs a leak. The knowledge of the leak is accepted by the system through the monitoring methodology. This methodology can be implemented using pressure sensors or temperature sensors or simple circuit breakers. On accepting the knowledge of leak in the system the system by itself stops the flow of oil in specified pipeline. It goes beyond this and finds out the next best route through which to send oil from the source to destination. This puts a stop to wastage of resource (oil in this case) and carries on the functioning of the system (if only a little less efficiently). Also since the system is real time the reaction time of the system to any such fault is very small and precise.

Thus this solution incorporates all major goals of the system specified above. Above this the system can be scaled on to larger systems. That is, if the oil distribution system grows the system can be mapped to the bigger distribution network. Again such a solution can be thought of for any network concerned with the flow of fluids such as water supply etc.

2. PROBLEM DEFINITION

The aim of this project is to develop a response system, which caters to the problems involved in transmission of fluids through big networks of pipelines. The main goals have been identified as minimization of resource wastage and lowering the detection-to-reaction time in case of a leak or other disruption in the network.
The stakeholders [6] in this situation are the organization that has the responsibility to manage and maintain the pipeline network, and the end users of the resource.
In our case the solution has to be developed with the following constrictions in view:

1. Financial constraint because of limited resources. We cannot acquire expensive equipment such as pressure sensors and electronically operated mechanical valves.
2. Man power constraint, as only two people have been involved in developing the solution.
3. Hardware constraint, as there is no physical network available on which to test the developed solution.
4. Time constraint of three months to develop the entire solution.

Keeping in mind the above constraints the problem definition gets transformed into the following:
There are three pipelines connecting a source and its destination. The pipelines have different delays associated with them, which the network will detect on it's own. This involved making the system test each line at the time of startup and thereby build a map of the network. The delays associated with each line also need to be calculated during this process. The source and destination should also be checked for being empty or overflowing respectively.
The flow of water or oil needs to be routed from the source to the destination with greatest efficiency. The system should also be fault tolerant in as much as it needs to detect any leaks and take appropriate steps in case of a leak. These steps should include stopping the flow through the line, which has broken down if it was in use. If the line wasn't in use the system should insure that the line would not be used in any subsequent routing decision.

All the above actions need to be carried out in real-time to satisfy the major goal of reducing the reaction time of the system. This means that the system cannot work on top of any present general-purpose operating system such as Windows and LINUX. Also, the system needs to be small and therefore cannot be made by customizing any existing general-purpose micro-kernels such as Neutrino micro kernel of QNX.

Therefore the development of a new micro-kernel by extension involved developing memory management modules, input/output modules, process management modules, controlling the timer interrupt and also providing means for reliable inter process communication.

The system, which is going to be developed should accept input from the keyboard and provide output through an interface with the parallel port. The status of the pipelines, the source and the destination is displayed through red and green light emitting diodes (LED’s). The system being developed at present caters to only three pipelines but once implemented can be scaled to a larger network.

4. IMPLEMENTATION

The entire design of the project has been implemented using 8086 assembly language. It could have been done using C, but using GCC would have created thirty-two bit codes, which would have been incompatible with 8086, as it is a sixteen-bit microprocessor. The major modules, which have been implemented, have been explained now.

4.1 Major structures:

The kernel of PIKACHU and the application revolve around the following structures:

1. PCB structure:

```
pid      equ 0           ; unique pid
ptype   equ 2            ;type of process- routing task etc(broad functionality)
priority equ 4           ;priority of proc
_AX equ 6                ;save registers-here using dos's inbuilt data type
_BX equ 8
_CX equ 10
_DX equ 12
_SI equ 14
_DI equ 16
pc equ 18                ;save prog. counter
_SP equ 20
_SS equ 22              ;save stack pointer& st.seg.
strt_time equ 24        ;when did process start
MAX_VAL equ 28                  ;value function
p_next equ 32            ;..
p_prev equ 36                   ;next & prev pointers for pcb list
stack_space equ 40;  ;where is stack stored in context switch
_CS equ 44
_DS equ 46              ; code seg, data seg
status equ 48           ;running or ready state
                        ; padding for future enhancement
                        ; total pcb-45 bytes
;size =50 hence we allocate 50 bytes at a time
```

2. Message structure:

```
msg_type equ 0          ;       1st bit-timer or kernel,
                        ;       2,3,4- kernel_msg subtypes
                        ;       0 in 1st bit means msg for timer
                        ;       2nd bit for id task,routing   ;
                                task,srcdst_chk,tester etc
                        ;       3rd for timer task
                        ;       4th from task exiting

line_id equ 2 ;         ;       for lines given by id task or
                        ;       routing-src,dest,line1,2,3
```

```
                                ;       this will also be used as the status when
the
                                ;       srcdst_chk task writes to kernel,

pid_from equ 4          ;process id of process to be woken if timer is sending to
                        ;kernel
                        ;or if msg ment for timer then this is the process
                        sending ;msg
                        ;or the status os line that has changed and has been
                        id'ed ;in line_id above is given
pid_to equ 6            ;only meant for timer- process to be woken(alarm set)
time_set_at equ 8       ;time at which alarm is set (in millisec)
delay equ 12            ;ticks to add to time_set_at(in msec) to wait
m_next equ 14
m_prev equ 18                   ; the next and previous pointers
                                ; total size of universal msg is 22 bytes

;size =22 hence we allocate 25 bytes at a time
```

3. Memory map structure:

```
line_no equu 0          ; link no
usable equ 2            ; can be used in future -0|1
used equ 4             ; is under use or not  -0|1
src equ 6
dst equ 8               ; src & dest.
cost equ 10             ; cost of line
                                ; total size of every entry- 5 bytes

;size =12
```

4. Process table structure:

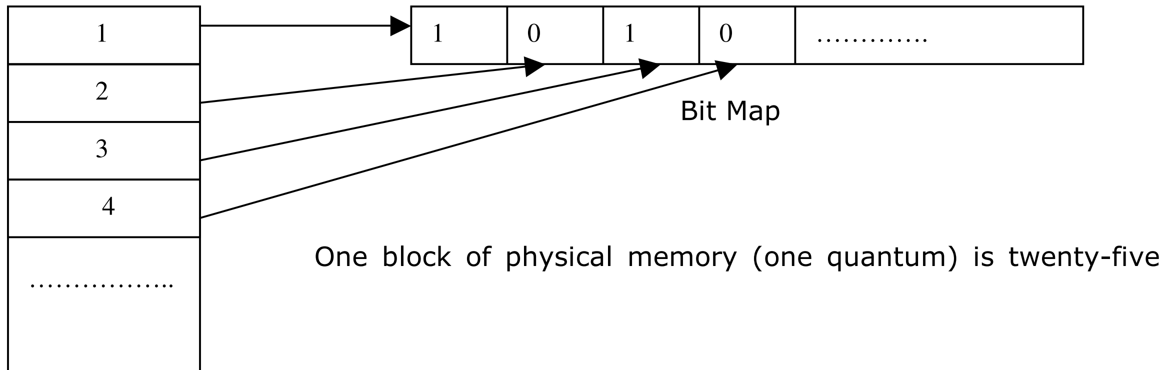
```
proc   equ 0                    ; process name
procid equ 2                    ; read and loaded from ids.h during table
creation
strt_addr equ 4                 ; starting addr of process
stack_addr equ 8                ; starting addr of stack
VAL_F equ 12;                   ;value function updated by timer
t_priority equ 14               ; priority

; total size of every entry- 20 bytes
```

4.2 Memory management module:

Memory is an important resource and needs to be carefully managed. The various schemes used for memory management can be based on bit maps, linked lists, paging, segmentation to name a few. The aim of the MMU is to provide a set of API's, which manage allocation, and de-allocation of memory in a safe and secure way.

The platform being built is a real time kernel and therefore the general safety measures and checks ensuring overlap protection should not be implemented [1]. The strategy selected for memory management is by bit maps.



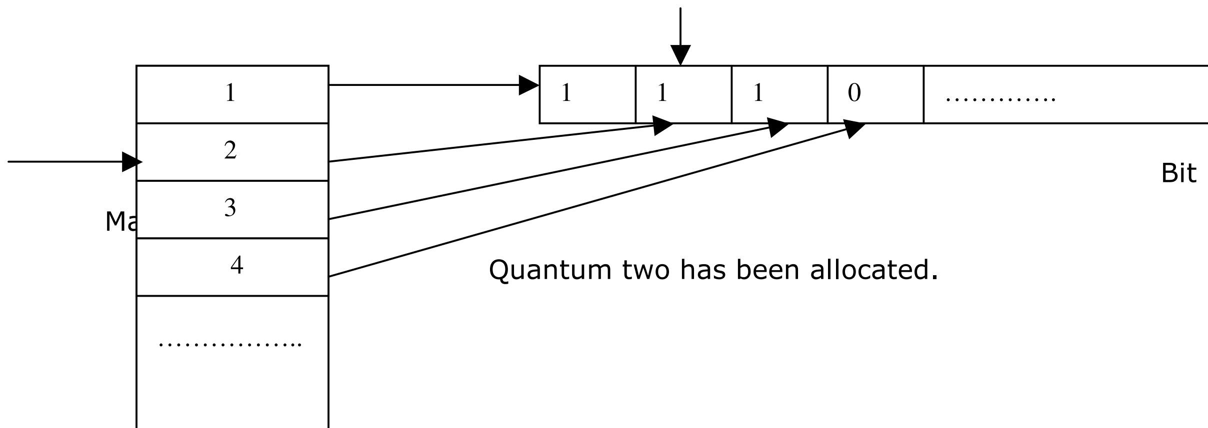
One block of physical memory (one quantum) is twenty-five

Physical Memory

Figure 4.1

When memory is assigned dynamically the operating system must manage it. With a bit map, memory is divided up into allocation units (quantum) of twenty-five bytes each. Corresponding to each quantum is a bit in the bit map, which is zero if the quantum is free and one if it is occupied. Figure 4.1 shows part of the memory and corresponding bit map. The size of the quantum is an important design issue, the smaller the quantum the larger the bit-map.

Imagine that the application requests one quantum to be allocated. Since the first bit in the bit map is set as one, it signifies that the first quantum is already occupied and therefore cannot be used. The MMU therefore must traverse the bit map till it encounters a bit which is zero. On finding such a bit the MMU sets the bit to one and returns the address of the newly allocated quantum. Figure 4.2 illustrates the process.



Quantum two has been allocated.

Physical Memory

Figure 4.2

To free a quantum the MMU should set the corresponding entry in the memory to zero. But care must be taken that this is done for the correct address and bit pair. Otherwise the application would end up using pre-allocated memory and therefore crash sooner than later.

The API's provided by the MMU in PIKACHU are:

1. ralloc(quantum size): This API allocates a quantum of memory and returns the address. The parameters can be MSG or PCB as these are the two quanta's supported by PIKACHU.
2. rfree(quantum size, address): This API de-allocates a quantum of memory. The parameters can be MSG or PCB as these are the two quanta's supported by PIKACHU, and the address of the memory quanta to be de-allocated.
3. getmap(): This API is used to return the central memory map of the network.

4.3 Timer module:

This module includes the timer interrupt handler and the extended features provided by the PIKACHU timer like the facility to set alarms etc. To achieve the said objectives we must load the interrupt service routine designed by us into the ISR vector table at the time of initialization of the system.

```
;;;;;;;;;;;load isr of timer;;;;;;
mov ax,0
mov es,ax
mov [ es:34 ],cs
mov ax,isrTimer
add word ax,0x600
mov [ es:32 ],ax
;;;;;;;;;;;;;;;;;;
```

Code to load ISR of timer

The timer chip 8254 works at a frequency of 18.2 Hz. This means that the processor will be interrupted 18.2 times per second. As we want the processor to work fast enough to complete one entire thread of execution without being interrupted, and still not ignore too many timer interrupts, we program the ISR to invoke the timer function once every 500 milliseconds.

The main job of the timer ISR therefore becomes the updating of the time counter variable and to check if the timer function needs to be invoked. If the timer function needs to be invoked, we have to jump to its code. But we also must make the ISR return as till the IRET instruction is not executed, all interrupts will remain disabled.

To achieve the jump to timer function and have IRET execute at the same time, we change the stack of the ISR and load the return address as the starting address of the timer function.

The API's provided by the timer module are:

1. gettime(): This API returns the value of the time counter variable sec_count. It can be used to get the number of seconds elapsed since the startup of the system.

4.4 Process management module:

This module implements basic process management routines. The PCB structure given in section 13.1 forms the basis of process management. The process to be run will be first created as a PCB. The following criteria should be considered before any design decisions are taken in the area of process management. As we are working towards a real-time system there are certain very specific constraints on each task. We have to have a general understanding of such constraints in order to deal with them.

Jobs and Processes:

For the purpose of describing and characterizing various types of real-time systems and methods for scheduling and resource management we call each unit of work that is scheduled and executed by the system a ***job***, and a set of related jobs which jointly provide some system function a ***task*** [2].

The ***release time*** of a job is the instant of time at which the job becomes available for execution. The job can be scheduled and executed at any time at or after its release time. The ***deadline*** [2] of the job is the instant of time by which it's execution is required to be completed. We will implicitly assume that the rate of progress a job makes towards its completion depends on the speed of the processor.

Temporal parameters of real-time workload:

Each job J_I is characterized by a set of temporal parameters, functional parameters, resource parameters and interconnection parameters. The temporal parameters tell us the timing and behavior [11].

Periodic tasks:

In a periodic task each computation or data transmission that is executed repeatedly at regular or semi regular time interval in order to provide a function of the system on a continuing basis is modeled as a task. In PIKACHU such tasks are the timer function being invoked one every 500 milliseconds, the scheduler being woken up in order to select the next task to be run form such periodic tasks.

Scheduling:

Jobs are scheduled and allocated resources according to a chosen set of scheduling algorithms and resource access control protocols. The module, which implements this algorithm, is called the scheduler. A scheduler provides only valid schedules, which satisfies the following conditions:

1. Every processor is assigned to at-most one job at anytime.
2. Every job is assigned at-most one processor at anytime.
3. No job is scheduled before its release time.
4. Depending on the scheduling algorithms used the total amount of processor time assigned to every job is equal to its maximum or actual execution time.
5. All precedence and resource usage constraints are satisfied.

In PIKACHU the scheduling we use is based on a priority driven approach.

Assigning the processor to a process:

The scheduler has the task of selecting the best available process for execution from the pool of jobs, which can be executed. But the scheduler does not do the task of assigning the processor to the selected process. It is don't by the dispatcher, in PIKACHU the pick_proc() function modeled on the MINIX pick_proc function [3]. The most important of the implementation of pick_proc is

```
;; making the chosen proc the running proc
mov word [running+2],bx
mov word [running],cx


mov si,bx
mov word ax,[ds:si+_CS]
mov word [jump_address],ax
mov word ax,[ds:si+_AX]
mov word bx,[ds:si+_BX]

;;;;;;;;;;;;;;;;;; CONTROL BEING GIVEN TO NEW PROCESS;;;;;;;;;;
```

```
jmp far [jump_address]
```

Code to assign the processor to a newly selected process

The API's provided by the process management module are:

1. create_pcb(who,param1,param2): This API is used to create a PCB for task ***who***. Any parameters or startup information to be passed to the task should be sent to this API through the two parameters ***param1*** and ***param2.*** These will be loaded into the registers AX and BX and then can be accessed by the newly created task when it runs.
2. ready_write_q(PCB): This API inserts the parameter PCB into the ready queue of the kernel. The address of the PCB should be passed not the constant PCB.
3. sched(): This API is not accessible to the application. It is only called by the kernel and forms a part of the kernel space of PIKACHU. It modifies the priorities of the processes in the ready queue and the running spot based on the amount of resources they use.
4. pick_proc(): This API is again a kernel space API and is not accessible to the application. It has the task of selecting the best available process from the job pool comprised by the ready queue and the running spot. The selected process is then assigned to the processor.

4.5 Inter Process Communication (IPC):

Various models can be used for IPC in an operating system, namely:

1. Message queues.
2. Mailboxes.
3. Pipes.
4. Semaphores.
5. Interrupts [3].

Though the last two are generally considered as being used for critical section problems and exception condition detection respectively, they can still be seen as a potential communication mechanisms [1]. In PIKACHU we make use of message queues because mailboxes are not very secure when we have to synchronize between various processes. Pipes also suffer from the disadvantage of taking too much time and that cannot be supported in a real-time system. Message queues on the other hand lend themselves to easy synchronization and faster implementation. Though there is a ready queue, it is not used for IPC. There are two queue's which

are used for the purpose if IPC. They are the kernel queue and the timer queue. Any process to communicate with the kernel uses the kernel queue. Referring to section 13.1 and the message structure, we find that the following types of messages are supported to communicate with the kernel:

1. Message from exiting task: This is a message from an exiting task like the light task. The number nine in the msg_type field of the message structure denotes it.
2. Message from a task returning a value: The tasks such as routing and idtask identify a line and wish to communicate with the kernel regarding this. They fill msg_type as three and the return value in pid_from field.
3. Message from timer: When a task sets an alarm and it expires, the timer communicates with the kernel using the msg_type five. It is a message asking the kernel to restart execution of the said task denoted by the pid_to field of the message structure.

The timer queue is a sorted queue. It is sorted on the time delay in the alarm values. A message of type one is sent to the timer queue with the pid_to and pid_from fields set to the process id's of the processes setting the alarm and the process to be woken up on expiry of the alarm respectively.

The following API's are provided to implement IPC in PIKACHU:

1. write_q(type of queue,address of msg): This is a generic API and can be used to write to both the timer and kernel queue's.
2. read_q(type of queue): This is a generic API and can be used to read from both the timer and kernel queue's.

4.6 Input/Output module:

The key concept in the design of the I/O software is known as device independence [3]. What this means is that we should be able to write programs without having to modify the programs for each different device types. Another important issue is error handling. In general errors should be handled as close to the hardware as possible. In PIKACHU the basic I/O consists of accepting input from the keyboard and giving output to the monitor and set of LED's interfaced through the parallel port. The keyboard handler should be loaded into the interrupt vector table at the time of initialization of the system.

```
;;;;;;;load isr of keyboard;;;;;;;;;;
mov ax,0
```

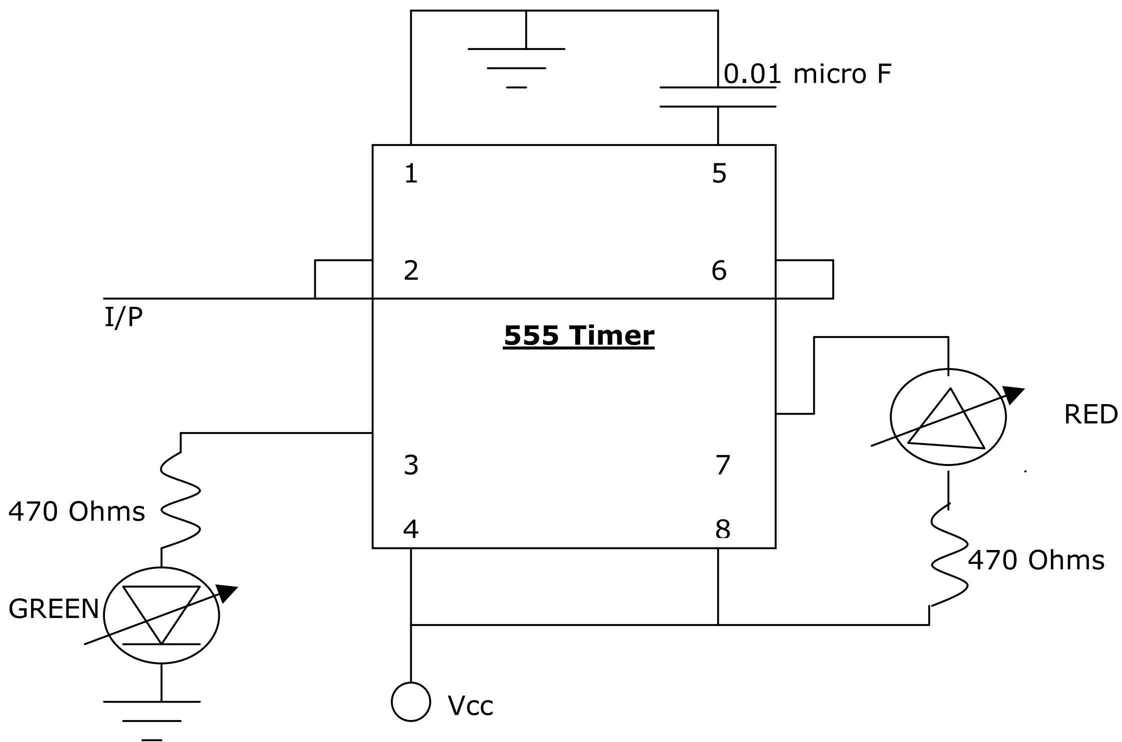
```
mov es,ax
mov [ es:38 ],cs
mov ax,kbd
add word ax,0x600
mov [ es:36 ],ax
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Code to load ISR of keyboard

The scheme used to write to the monitor is to write to the video memory buffer of the system. This buffer starts at segment address 0xB800 and the offset signifies the cursor position. Care must be taken because every single position is denoted by two bytes, one for the character to be printed and one for the attribute such as back ground color etc.

Interfacing with the parallel port is a very simple procedure as the parallel port stores the output value in a buffer and maintains the voltage levels at the port till the next set of bits is written to the buffer. This removes the necessity to keep refreshing the port. The output is given eight bits at a time. A simple circuit, which uses 555-timer chip to toggle the voltage across the two LED's, is used.

Figure 4.4 Circuit diagram for LED display



4.7 Tools used for development

The coding has been done in 8086 assembly language and has been carried out on the LINUX platform. Various tools that have been used are:

- **NASM:** The Netwide Assembler, NASM, is an 80x86 assembler designed for portability and modularity. It supports a range of object file formats, including Linux `a.out' and ELF, NetBSD/FreeBSD, COFF, Microsoft 16-bit OBJ and Win32. It will also output plain binary files. Its syntax is designed to be simple and easy to understand, similar to Intel's but less complex. It supports Pentium, P6 and MMX opcodes, and has macro capability.
- **NDISASM: T**he Netwide Disassembler - 80x86 binary file disassembler. It supports various options for disassembly such as a sixteen-bit and thirty-two bit disassembly modes.
- **MCEDIT:** This is a very powerful text editor provided by LINUX and has been used to edit the code.
- **DD COMMAND:** This is a very powerful system command implemented under LINUX, which allows code to be written directly onto the specified sectors of the floppy.

6. FUTURE ENHANCEMENT

The system implemented in this project is a simple system, which provides a solution for a network consisting of one source and one destination. The system has been kept simple in order to express the central idea behind it, which we believe is more important. Hence we can visualize this system being mapped on to much bigger and more complex networks which should be done very easily.

A number of features such as "Integration" of a new line by the administrator can be implemented in the future. If sufficient resources are present the detection of flow and any fluctuation in the same can be detected with better accuracy using different means of monitoring fluid flow.

The kernel of PIKACHU can be further developed into a more generic kernel. This would then be helpful in creating similar such applications for different kinds of networks such as networked power grids, natural gas pipelines etc.

A better interface would attract more users to it. But while developing it must be kept in mind that the main quality of PIKACHU is not to look good but to act powerful. This means, that an enhanced user interface should not come at the cost of reduced performance of the kernel.

Actual mechanical valves could however replace the LED's, that have served us so well, so that fluid flow is shut down or carried on depending on whether the valves are shut or open (RED or GREEN for LED's).

As PIKACHU is transported to faster and faster architectures small modifications such as better interrupt routines for keyboard can be used without much of a problem.

We hope someone carries on our work and this solution can actually go on to fulfill its destiny.