http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 6 (3): 376-379, 2007
ISSN 1812-5638

Real-Time Scheduler for Transport Protocols

Samia Aslam Sherwani and Malik Sikander Hayat Khiyal
Department of Computer Science, Faculty of Applied Sciences,
International Islamic University, Islamabad, Pakistan

Abstract: Real-Time Operating Systems (RTOS) currently available in industry, for embedded systems, require multitasking support in the targeted processor. The category of such operating systems is known as Pre-emptive Multitasking Kernels. However multitasking support is not provided by all processors. We have developed multitasking scheduling technique (Collaborative Multitasking) for the processors or microcontrollers which do not have built-in multitasking support such as support for context switching, for example, 89C51 Microcontroller and 89C52.

Key words: Collaborative multitasking, context switching, embedded systems, microcontrollers, real time operating system, real time scheduling, TCPIP suite

INTRODUCTION

In the modern age, the intelligence of computing power has been integrated into every device and gadget, resulting in embedded systems. An embedded system refers to a device with computer logic on a chip inside it, typically consists of a single-board microcomputer with software in ROM designed to perform a dedicated function. Such systems are structured in a different manner as compared to high performance desktop systems. Their designing issues include: Low cost, predictability, responsiveness (Seo *et al*., 1998) and temporal accuracy (Kopetz and Octsenreiter, 1987).

Embedded systems normally exist as part of a bigger system and are constructed with the least powerful processors that can meet the basic functional and performance requirements so that the manufacturing cost of the equipment can be lowered. As discussed by Agarwal and Bhatt (2004), due to absence of general features and extremely tight design constraints, unlike in conventional systems; the developers of embedded systems have to work with complex algorithms to manage resources in the most optimized manner.

Scheduling is a mechanism that determines which job has to be executed from the pool of jobs in system on the basis of the scheduling algorithm implemented. Whenever multiple tasks share common processing resources, they require their states to be stored at the time of process switching, so that these can be restored afterwards. The state includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary. Often, all the data that is necessary for state is stored in one data structure, called process control block. According to Nacul and Givargis (2005), in order to support multitasking on a system, an operating system layer is needed, which it is not commonly available in embedded systems due to lack of sufficient memory. Examples are PIC (Huang, 2005) by Microchip and 8051 (Calcutt *et al*., 1998) by Philips. These microcontrollers are cheap enough to give cost effective devices. If these micro-controllers are being planned to be utilized for handling complex multitasking scenarios, it is only possible if handled programmatically within embedded software design.

REAL-TIME SCHEDULER: COMMON PRACTICE

Donald Gillies defined real-time system as follows:

- A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced.

Real-time systems perform a number of tasks at a time. The Resource Manager allocates processor time to each task according to the schedule in such a way that the tasks appear to be parallel. The process of choosing a task to grant resources to, that is, Central Processing Unit time is called Real-Time Scheduling as defined by Liu and Lee (2003).

Corresponding Author: Samia Aslam Sherwani, Department of Computer Science, Faculty of Applied Sciences, International Islamic University, Islamabad, Pakistan Tel: (92) 03345046539

Preemptive multitasking: A typical strategy to implement real-time scheduling is called preemptive multitasking. In preemptive multitasking, operating system uses some criteria to decide how long to allocate to any one task before giving another task a turn to use the operating system. The act of taking control of the operating system from one task and giving it to another task is called preempting.

To perform resource management using preemptive multitasking, the resource manager has to perform two duties:

- Context Switching
- Task Scheduling

When a task is in running state and the time slice has been expired such as timer event, the scheduler is invoked which decides which task deserves to be given next time slice using its scheduling algorithm based on priority system. Finally, it performs context switching, replaces first task by second task and lets the later task to perform its duty. Figure 1 explains the scenario.

Limitations in preemptive multitasking: Preemptive multitasking is implemented on a processor or a micro-controller, which has built in support for context switching and a periodic task trigger on which event scheduler has to be invoked. A common criterion is simply elapsed time: the timer implemented in hardware is programmed to be invoked on expiration of a time slice. The timer generates an interrupt, which initiates an interrupt service routine. In interrupt service routine, scheduling is performed and it is decided which task is to be granted processor next. The state of currently executing task is saved and the context of the next task is loaded into the CPU. After ISR, the CPU starts executing the newly loaded task. So, to perform task switching, the CPU must have spare context registers, called 'Register Banks' as shown in Fig. 2.

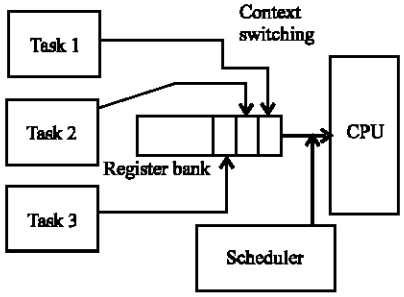

Fig. 1: Real time scheduling

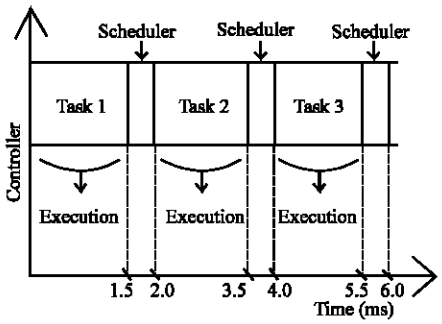

Fig. 2: Preemptive multitasking

High level processors, such as, Intel 8086, Intel 8088, Intel 80386, Intel 80486, Pentium and Pentium Pro support pre-emption. There are number of micro-controllers that provide built in hardware support for context switching and periodic task trigger, for example C166 and C167 by Siemens (2000).

COLLABORATIVE MULTITASKING APPROACH

As explained earlier, the multitasking performed by context switching requires very particular hardware support, which is not available in tiny micro-controllers such as Intel 80C51 and 80C52.

This article gives a programming model to implement multitasking in real-time tasks, for example, running a TCP/IP based application. Collaborative multitasking model gives the idea to address fundamental issues of running preemptive multitasking kernel on tiny micro-controllers.

Tasks collaboration: In collaborative multitasking, tasks (any user process running on that controller) collaborate with each other in a way that each task executes a part of its route, saves its state locally and then releases system resources voluntarily.

In this system, each task is represented by function or routine. In this idea, no task is forced to preempt resources from it. A task returns after executing a part of it, saves its state and gives control to other task waiting for resources as shown in Fig. 3. The sequence executes in a continuous fashion.

For example, we have three tasks. First Task is Display task whose responsibility is control LCD display. Second task is Comm task whose responsibility is to receive any data from comport and process it and the third one is KeyPad task which scans the keys and gets any activity of key pressing. Now these three tasks will collaborate with each other. When Display function will be called, it will scan all the display memory and will refresh it on the screen in one cycle. After that it will

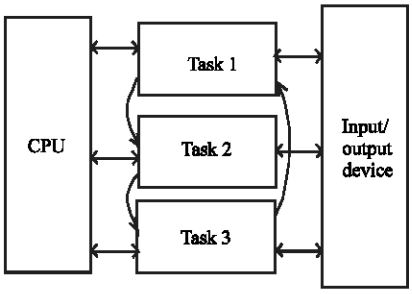

Fig. 3: Collaborative multitasking

return back and Comm task will be invoked. In a single cycle, Comm will scan its comport, receive any incoming waiting data and process it.

After that it will return back and then finally KeyPad task will be invoked. In a cycle, KeyPad will scan all the keys and will refresh keypad memory indicating any key press event. This sequence will execute continuously.

```
main ()
{  InitSys ( );
   While (1)
    {        Display ( );
             Comm ( );
             KeyPad ( );
   }
   QuitSys ( );
}
```

The sharing of resources among the tasks is not based on time slices, but sharing is done on work basis or number of instructions. Every task divides its whole work into sub-tasks. Whenever a task is given control of CPU, it executes one of its sub-tasks and returns the control. In next allocation of CPU, it executes next sub-task.

For example, we have an embedded system which has to execute three tasks: Task 1, Task 2 and Task 3 simultaneously. Task 1 is further divided into three subtasks: subtask 1, subtask 2 and subtask 3. Task 1 completes, as each subtask executes ones.

```
 While (1)
 {        Task1 ();
          Task2 ();
          Task3 ();
 }
 Task1 ()
 {        static int nStat=0;
          switch (nStat)
          {  case 0: Subtask1(); nStat=1; break;
          case 1: Subtask2(); nStat=2; break;
          case 2: Subtask3(); nStat=0; break;
          }
}
```

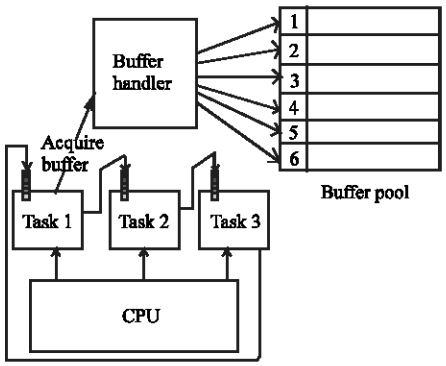

Fig. 4: Inter-process communication

The scheduler is designed such that every task executes its one sub-task in its turn and returns back so that next task can be executed. In above example, Task 1 completes in three iterations. In this way, all the tasks are executed simultaneously because of their collaboration with each other.

Queues for inter-process communication: Inter-process communication is always an important issue when designing scheduler for real-time embedded systems. In collaborative multitasking programming model, every task has its incoming and outgoing FIFO and also there is a shared buffer pool. Whenever a task wants to send data to another task, it acquires a free buffer from buffer pool, copies the data in buffer and puts the index of buffer in incoming FIFO of their task. Every task polls its incoming FIFO and processes the data, if present, as shown in Fig. 4.

Task priorities: Task priority is very important concept in multitasking system. The priority represents the relative importance of a task at run time. When three tasks are running at a time, then the process of determining which task deserves CPU more is called priority.

For example, we have Task 3 that is more important than Task 1 and Task 2. Then its priority can be implemented, as its iterations can be increased relative to other tasks.

```
While ()
{
Task1 (4);
Task2 (1);
 Task3 (2);
 I/OTask (1);
}
void Task1 (int nPriority)
{
```

```
int nIteration=0;
While (nIteration<nPriority)
{
//execute subtasks
nIteration++;
}
}
```

The priority of a task can be determined at run-time and it can be set according to the situation.

Example: Here, we describe the design and development of real-time scheduler for transport protocols, such as TCP/IP. The TCP/IP protocol stack is implemented such that each layer is represented by a task.

```
INIT_STACK;
 WHILE (nQUIT)
 {
 APP_TASK;
 TCP_TASK;
 UDP_TASK;
 IP_TASK;
 PPP_TASK;
 COM_TASK;
}
```

The main thread initializes all the layers, distributes the time slices by calling respective processes. Each layer works in two directions, that is, it processes data from upper as well as lower layer. A separate buffer bank is reserved for data to be processed, in the form of two-dimensional array. Each buffer has following associated attributes:

- Name of the buffer (Free, Temporary, PPP Down, PPP Up, IP Down, IP Up, UDP Up, UDP Down, TCP Up, TCP Down, Application Down, COM Up)
- Command (No command, dial, ping, valid IP frame, etc)

Message flags associated with each process control sub-processes. Each layer has two Data Queues associated with it, one for each direction: Layer up Job Queue, that contains pointer to the buffer received from upper layer and is ready to be processed according to the command associated with the buffer and status of the message flag associated with that direction and Layer down Job Queue, for data received from down layer. These Job Queues are responsible for inter-process communication.

Whenever an application wants to perform a TCP/IP related task, it gets a buffer from buffer bank, adds data to buffer, associates a command with buffer which indicates what has to be done with the data in buffer and passes buffer reference to the Layer Up Job Queue of the lower layer.

On turn of task associated with the next layer, the incoming job queue is checked and the buffer is processed according to the command, flags are set and the buffer reference is added to Layer Up Job Queue of the next layer. Next layer behaves in same way, until data reaches COM layer and is written to COM port.

It is not necessary for a task to complete its job in single iteration. So, each task has to maintain its state, so that it can continue from the same point in next iteration. For that, each layer performs part of its task, saves its state in buffer and keeps the track of previous work with the help of flags associated with each task.

CONCLUSIONS

This TCP/IP stack is tested with SMTP/POP3 application and also compiled on Keil Cross compiler for embedded system support. As the whole stack works in one thread, so it does not require multitasking support in target processor. Hence the research is successful.

REFERENCES

Agarwal, S. and P. Bhatt, 2004. Real-time Embedded Software Systems: An Introduction. Technology Review.

Calcutt, D., F. Cowan and H. Parchizadeh, 1998. 8051 Microcontrollers: Hardware, Software and Applications. John Wiley and Sons, New York, USA.

Huang, H., 2005. PIC Microcontroller: An Introduction to Software and Hardware Interfacing. Delmar Learning, New York, USA.

Kopetz, H. and W. Ochsenreiter, 1987. Clock synchronization in distributed-real-time systems. IEEE Transactions on Computers, C-36: 933-939.

Liu, J. and E.A. Lee, 2003. Timed multitasking for real-time embedded software. IEEE Control Systems Magazine, 23: 65-75.

Nacul, A.C. and T. Givargis, 2005. Lightweight Multitasking Support for Embedded Systems Using the Phantom Serializing Compiler. Proceedings of the Conference on Design, Automation and Test in Europe, 2: 742-747.

Seo, Y., J. Park and S. Hong, 1998. Efficient User-Level I/O in the ARX Real-Time Operating System, Lecture Notes in Computer Science, Vol. 1474.

Siemens, 2000. C167CR/C167SR 16-Bit Single Chip Micro-controller, Data Sheet, V3.1.