Embedded Systems with Improved Interprocess Communication Design

Hosein Marzi
Department of Information Systems
St. Francis Xavier University
Antigonish NS B2G 2W5 Canada
hmarzi@stfx.ca

Larry Hughes
Department of Electrical and Computer Engineering, Dalhousie University,
Halifax NS B3J 2X4
Canada
lhughes2@dal.ca

Yanting Lin
Department of Electrical and Computer Engineering, Dalhousie University,
Halifax NS B3J 2X4
Canada

Abstract-This paper describes a new approach in designing communications between processes in a multi-threaded environment. This technique supports real-time performance and can be adapted for embedded operating systems. At present, Message-Passing is a widely used technique in designing Interprocess Communication Systems. This design may degrade performance of Real-time Systems. The present research introduces a library-based architecture for Interprocess Communication Systems (IPC) adaptable in Real-time Embedded Operating Systems. It achieves improved real-time performance by running IPC as a set of library functions.

I. Introduction

The designers of the first time-sharing operating system CTSS and then Multics coined the term Process in mid 60s. [1]. In the original glossary of Multics a process is described as an address space. In the current context it is referred to a program in execution and may be used, to some extent, interchangeably with 'task'. Processes can be isolated entities or may cooperate to accomplish a common objective as part of a cluster of processes. Particularly, in systems supporting concurrent programming and distributed environments, processes can communicate with other processes. Cooperating processes might need to synchronize their activities or exchange data among themselves. An Interprocess Communication System is the mechanism which provides this medium.

LINUX operating system uses signals, pipes, and most commonly message-based IPC. IPC techniques used in Windows include data-oriented IPCs which are traditional UNIX techniques such as pipes, message queues, referred as mailslots, and shared memory. Windows also uses procedure-oriented or object-oriented techniques using local or remote procedure calls, LPC, RPC, and Component Object Model, COM. LINUX and Windows operating systems are not designed for real-time applications. Processors functioning under non-real-time operating systems allocate time slices to all processes fairly to improve average performance. A distinct feature of real-time operating systems is in the precise timing and predictable performance. A real-time system is defined as "a system in which the correctness of the computation not only depends upon the logical correctness of the computation but also upon the time at which the result is produced" [2]. Special purpose microprocessors functioning in embedded systems are selected to just adequately respond to the requirement of the particular system. An embedded system may run under a real-time or non-real-time operating system. Some operating systems develop their own simplified IPCs, such as MINIX [3]. Current designs for real-time operating systems adapt either non-POSIX IPCs, example of which is RT-LINUX [4] that uses proprietary application program interfaces, or adhere to message-based architecture and use pure message passing mechanism in the cost of performance, QNX [5]. In message-based designs, system services and process communications run as processes. The steps required to support these system calls involve in time-consuming process switching; which degrades the real-time performance [6]. The current study focuses on developing a real-time compatible IPC system with library-based architecture to operate in embedded systems. The IPC is POSIX compatible and a mutual exclusion mechanism using an atomic test-and-set

978-1-4244-3760-3/09/$25.00 © 2009 IEEE

instruction enhances the real-time performance of the operating system [6]. IPC runs as a set of library functions, in this new approach as a set of library functions, instead of sending messages.

II. IPC System Design

This section describes the design of a POSIX compliant IPC system for real-time embedded systems incorporating Motorola MC68HC12 microprocessor. The proposed IPC system utilizes the related data structures in the user process and file system. This support is required on two grounds. First, each IPC object is associated with a file name; this allows the name-handling feature of the file system to be used for IPC, eliminating the need for creating an additional IPC naming system.
Second, as most IPC system calls need descriptors, these are created and maintained in the user process space as input parameters to reference corresponding IPC objects.

An efficient message queue mechanism is used for exchanging messages between different processes and the semaphore, which provides a protection scheme for the shared resources. The message queue and semaphore have different features and purposes. However, they share common structures, including the magic number, free list, lock, and waiting queue. Figure 1 displays the structure of a user process containing the necessary fields of the real-time interprocess communication as well as the normal process control block PCB. In this design, IPC fields are not embedded into the PCB and IPC is independent of the kernel design. Therefore, IPC and kernel can be designed in parallel without interfering with each other.

Description of each field of the IPC structure is as follow:

pcb is a process control block structure designed by the kernel developer. magic is initially checked by every operation on RT_IPC_PCB to ensure it is working on the correct structure. The semaphore and the message queue have a waiting queue and each node of the queue is a Proc_Entry* blk_headptr structure containing a previous and next pointer to the same structure as well as a PCB pointer. blk holds a Proc_Entry structure and is used to link its own process with other processes when blocked. Some IPC operations deal with descriptors. A descriptor table, des_tbl[], pointed by des_tbl_ptr pointer, is established to maintain related information associated with descriptors in process space. The structure of a descriptor table, des_tbl[], is also shown in Figure 1. The descriptor table, des_tbl[], has four fields; des_type, oflag, fs_id, ipc_ptr, and are used for both regular files and IPC objects that are semaphore or message queue.

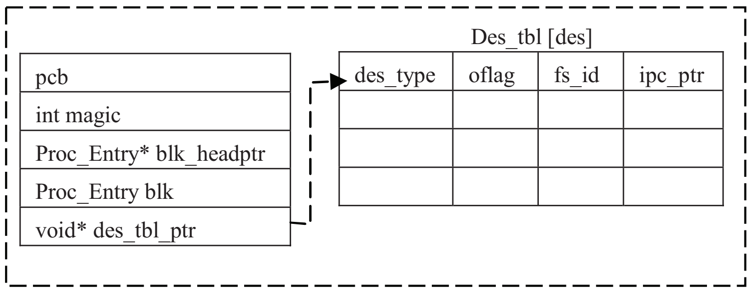

Fig. 1. RT-IPC-PCB

III. Disadvantages of the Current IPC Designs

The IPC system has modular design and its modules include the common IPC structure, the data structure of semaphore sem_t, the data structure of message queue mq_t, and the algorithm of IPC operation. The modular IPC system has the advantage of portability and is independent from the operating system architecture. The most import advantage of this modality, however, is the option of designing a message-based IPC or a library-based IPC without affecting the other individual modules.

In a message-based architecture, system services including the file system, memory management and device driver run as processes and message passing is the main method of communication for system services and user process communication. System services running as processes are referred to as system processes and have higher priority than user processes. If a user process requests a system service, kernel must switch the system and user process back and forth to finish a single system call. This is a practical method in most general purpose operating systems; however, it is not compatible with real-time system objectives.

Context switching which is switching from one process to another requires that the current process information (process state) available on the CPU registers be saved prior to entering into ready status from running state. On a MC68HC12, a voluntary context switching using software interrupt instruction (SWI) consume fourteen machine cycles. On the same processor, restarting the process using return from interrupt instruction (RTI) takes twelve machine cycles [7]. A message-based system requires two context switches.

The first one is for sending a message from requesting process and the second one from the responding process. This takes at least a total of fifty two machine cycles.

In the proposed Library-based IPC system, all system services including IPC, File System, device drivers, and memory managers are implemented as library function.

IV. Improving IPC Design for Real-time Operating Systems

In order to avoid disadvantages of message-based IPC, Library-based architecture is considered as an appropriate option. In a Library-based IPC, as its name implies system services, such as file system, memory management, device driver, naming system, as well as IPC are designed as library functions. In this approach a system call can be accomplished by a user process calling library functions. In this system, the process disables interrupts before accessing any library functions and enables them afterwards. This process requires a total of four machine cycles [7]., two machine cycles for setting interrupt mask, SEI and two for clear interrupt mask, CLI.

A library-based system has three layers. As shown in Figure 2, the first layer is kernel and consists of interrupt service routines, kernel semaphores and scheduler. The second layer is the system services implemented by library functions. The third layer is user processes. According to the Figure 2, the IPC system and file system are library functions and their computation takes the time quantum of the calling user process. User processes can directly call library functions that support system services. System services do not require process switching and message passing between system processed and user processes. Therefore a library-based system operates significantly faster than the message-based IPC system.

V. Avoiding Simultaneous Access to the Shared Structures

It is a requirement of real-time systems that the highest priority process can start running in a small bounded time [8]. A mutual exclusion mechanism prevents simultaneous access to the shared structures by more than one process. A mutual exclusion mechanism may result in a priority inversion, that is, a higher priority process waiting for a lower priority process owning the resource to release it. In order to avoid this problem and implement the mutual exclusion mechanism three possible solutions; Use of priority inheritance, Use of a test-and-set instruction and disabling interrupts, and Use of atomic test-and-set instruction, are reviewed and the last one was considered as the best solution. In the atomic test-and-set instruction approach, mutual exclusion is provided by reading, modifying and storing the lock value in one single uninterrupted instruction. Some processors, such as SPARC [9], provide an atomic method for test-and-set instruction method. Motorola 68HC12 does not provide this mechanism. Therefore an algorithm is designed to simulate the atomic test-and-set instruction for 68HC12 processors and has been used in the design of mutual exclusion mechanism.

VI. Implementation and Performance Measures

The test bed for the proposed IPC is a Motorola MC68HC12 microprocessor. This is a 16-bit microprocessor, which is upward code compatible with its predecessor 8-bit MC68HC11 [10], [11]. Algorithm of the IPC system is developed using C programming. The program structure include four sets of header files and source code files errno, ipc, mqueue, semaphore for errors, common functions and kernel functions, message queue functions and semaphore functions, respectively.
For example, mqueue.c file includes: message queue functions such as sys_mq_init(), sys_mq_destroy(), default_mqattr(), gen_new_msg(), enqueue_msg(), dequeue_msg(), sys_mq_send(), sys_mq_receive(),

User Process	Processes				
Library Function	IPC	Naming System	Device Drivers	Memory Management	File system
Kernel	ISR		Kernel Semaphore		
			Scheduler		

Fig. 2: System Architecture of the Library-based IPC

mq_getattr(), mq_setattr(). The name of each function is an indication of its role.

In order to study performance of the designed IPC, the files have been compiled and a number of tests have been conducted, including free list, waiting queue and lock queue of common IPC, message queue and semaphores. The designed IPC is modular and its performance can be tested independent from the rest of the operating system. However, the name handling system of the file system has been adapted in the IPC design. There are some POSIX IPC functions that use filenames and descriptors as their input arguments. In the test version of the current IPC functions, descriptors and names were replace by the IPC object pointers in order to make it possible to test the IPC independent of file system. In the actual system, IPC functions such as mq_open(), mq_close(), mq_uplink(), sem_open, sem_close(), sem_uplink() are implemented in file system with regular file system functions such as: open(), close(), and unlik().

VII. Conclusion

The research introduced the design and implementation of a library-based Interprocess Communication for real-time embedded POSIX compliant operating systems to suit Motorola MC68HC12. Current designs of IPC systems have message-based architecture or develop proprietary application program interfaces.

The paper elaborates on requirements of a real-time environment in an embedded system, and distinguishes this from non-real-time general-purpose applications. The article uses POSIX functions for semaphore and message queues as tools in designing the IPC. The proposed design benefited from name handling mechanism within the file system and also IPC system calls descriptors were created and maintained in the user process space. It was also emphasized that none of the fields of the designed IPC was embedded into the process control block, consequently the proposed IPC was independent of the kernel design and if necessary can be designed in parallel with the kernel.

Among three possible solutions for inclusion of mutual exclusion mechanism atomic test-and-set instruction was selected as a better solution which could assure the highest priority process can run in a small bounded time and prevent simultaneous access to the shared structures by more than one process.

The proposed library-based IPC system incorporated the aforementioned features could provide an open, efficient and POSIX compliant system. The system services were not implemented as processes but as a set of library functions. In the event of calling an IPC application program interface function, library functions were executed and the calling process was kept in the running state. Consequently, the overall real-time system performance was improved by avoiding processes switching. The system performance and feasibility of the design were verified at the system implementation phase and a number of tests including free list, waiting queue, message queue and semaphore were authenticated the designed IPC system.

References

1 Van Vleck T., *Multics*, General Information, http://www.multicians.org/general.hml, (Feb, 2005).

2 IEEE Standards, for IT POSIX Part 1:System API – Amendment 1: Real-time Extension [C language], *IEEE Standard*, 1003.1b, (1993).

3 Tanenbaum A.S., Woodhull A.S., *Operating Systems: Design and Implementation*, 2nd Ed., (Prentice Hall, NJ, 1997).

4 Barabanov M., *A Linux-based Real-time OS*, MSc Thesis, (New Mexico Institute of Mining & Technology, 1997).

5 QNX Neutron System Architecture, *QNX Neutrino System Architecture,* (July 2004), http://www.qnx.com/developers/docs/momentics621_docs/momentics/index.html.

6 Lin Y., *A Posix Real-time Interprocess Communication System for the Motorola 68HC12 Microprocessor*, MSC Thesis, (ECE, Dalhousie University, Canada, April 2003).

7 Motorola, *MC68HC12 Reference Manual*, (Aug 2004) http://e-www.motorola.com.

8 Sha L., Rajkumar R., and Lehoczky J. P., Priority Inheritance Protocols: An Approach to RT Synchronization; *IEEE Tran. on Computers,* Vol. 39, No. 9, (Nov 1990), pp. 1175-1184.

9 Maruo J., and McDougall R., *Solaries Internals*, (Prentice Hall Inc., NJ, 2000).

10 Lavandier J, Viot G., Migrating to the 68HC12 in C, (Feb 2005), Online document: http://www.enseirb.fr/~kadionik/68hc12/migrating_to_68HC12.pdf

11 Motorola, *MC68HC812A4 Advanced Information*, (Feb 2005), http://e-www.motorola.com.