A MULTI-MICROPROCESSOR ARCHITECTURE WITH HARDWARE SUPPORT FOR COMMUNICATION AND SCHEDULING

Sudhir R. Ahuja
Abhaya Asthana

Bell Laboratories
Holmdel, New Jersey 07733

ABSTRACT

We describe a multiprocessor system that attempts to enhance the system performance by incorporating into its architecture a number of key operating system concepts. In particular:

— the scheduling and synchronization of concurrent activities are built in at the hardware level,

— the interprocess communication functions are performed in hardware, and,

— a coupling between the scheduling and communication functions is provided which allows efficient implementation of parallel systems that is precluded when the scheduling and communication functions are realized in software.

1. INTRODUCTION

We describe a multiprocessor architecture that incorporates hardware support for many key operating system concepts. The architecture is extensible and is oriented towards supporting distributed computing. Such an architecture would be effective as a server node in a larger distributed system. The architecture is oriented towards microcomputers, especially Single Board Computers (SBC's). The intent here is to simplify the operating system functions executed in each SBC by providing hardware support for communication and scheduling.

The major functions of a conventional timeshared operating system are:

— memory management

— scheduling

— internal and external communication.

In single CPU systems a significant amount of processing time is consumed by these functions. This effectively lowers the processing bandwidth of the processor. The communication and scheduling problems are magnified even more in distributed systems. This further results in a bad mismatch between the communication capacity of the interconnection network and the processing capacities of the individual processors.

Figure 1 illustrates the inner loop of a typical multi-tasking operating system. Whenever an activity blocks, either voluntarily or due to an interrupt, its state is saved. The appropriate system function (for example a system call or interrupt handling) is performed. Upon completion the system decides which activity will next gain control of the CPU. The state of the selected activity is restored and it is allowed to resume execution. In our architecture, we separate the communication and scheduling functions from the data processing function as indicated by the dashed line in the figure. Further, we provide effective hardware support for the communication and scheduling within a processing node, thus freeing the processor to handle data processing.

The main feature of our architecture is that it realizes and supports the intimate coupling between communication and scheduling. Hardware support for communication functions have been suggested previously [RIC81], [DIT81], [SWA77]. This generally results in the speed up of low level communication functions. However, since these systems consider communication independently of scheduling, the improvement in overall system performance is limited.

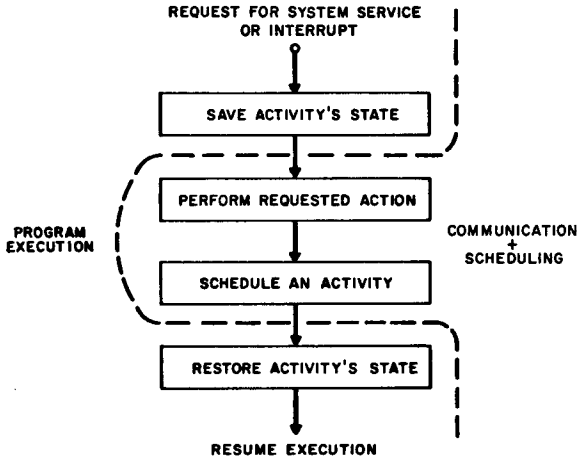


FIGURE 1 CORE OF A MULTITASKING SYSTEM

The architecture of our Processing Node (PN) is organized around a multi-microprocessor system based on shared memory as shown in Figure 2. The processors execute programs resident in their local memories. The shared memory is used only for efficient sharing of data structures. This partitioning is done primarily to reduce interference on the system bus [SMI77]. However, in certain cases, access to the shared memory may still result in significant amount of interference. The signaling, including synchronization, is accomplished through special hardware queues. The Signaling and Scheduling Processor (SSP) manages all the internal and external communication. In addition, it also handles the scheduling and binding of activities to Execution Units (EU). Section 3 describes the architecture in more detail.

2. MODEL

Our model of distributed computation consists of "server" and "client" processes. A server provides advertised services to client processes that connect to it. A common way of implementing a server is to create on demand multiple threads of execution, where each thread corresponds to an activity that handles a single client. A server process connects to the outside world via channels and communicates to it by sending and receiving messages over channels [AST81]. Messages are either requests to perform some action or are responses containing the results of an action. Requests are identified by a token. This token is carried by the request as it hops from one server to another in the process of completing itself. Eventually, a response message with the same token is received by the originating client or server process.

A server process runs on a PN. A process comprises many 'activities'. A message received by a PN is responded to by an activity. Many activities may be in progress at a given time.

Request Tokens

The messages received by a node are identified by a token number (t#) and a message type. These tokens are the similar to activation tokens in other systems [DEN80, ARV81]. From a communication perspective one can view them as a means of further multiplexing channels into finer threads of communication that allow activities within one node to be dynamically associated with activities in another node.

Channels

All input/output to external processes is accomplished using channels (c#). Channels provide bi-directional communication paths between processes. Channels are setup before being used and taken down when no longer required. Channels are owned by a server process and are shared by the activities comprising that process.

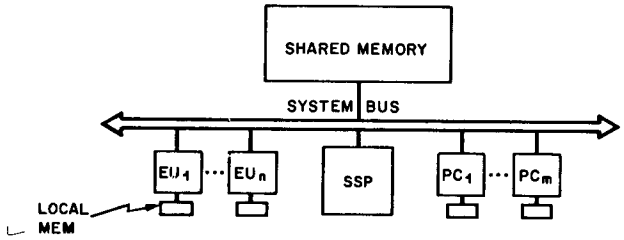
Activities

An activity is an instance of a procedure that interprets and acts upon messages received by a node. An activity has its own locus of control, a program to execute, and its own private data. All activities within a process share a common address space. Activities communicate and synchronize with the outside world via channels using Send/Receive primitives such as in [HOA78] or [CHE79]. Within a node activities may share data and synchronize with each other using semaphores [DIJ68]. The *binding* between a request token and an activity takes place when a new request token arrives and is *maintained* until that request is completely serviced. In order to process a request an activity may have to generate requests to other servers. An activity blocks whenever it issues a communication request. An activity is identified by its pointer (a#) which references an activity object that specifies the text for the activity, its private data and stack area and other state information related to that activity.

Activities execute on processors. The *binding* between an activity and a processor is *dynamic*. An activity that is ready for execution is assigned a processor on which it can execute so as to balance the load on all processors. An activity gives up a processor as soon as it blocks.

Activity States

The procedure for an activity consists of a finite number of steps that are executed sequentially. These steps may be pure data manipulation operations or may be synchronization operations. Any synchronization operation causes the activity to block until that synchronization is achieved. An activity, therefore, has three states. An activity may be in a "running" state. There is only one such activity in a processor at any given time. As soon as an activity issues a synchronization primitive it is suspended and gets queued on a "wait" queue. The entry on the wait queue is a pointer to a synchronization object that specifies the activity requesting the synchronization, the type of synchronization and other related data. When the desired synchronization is achieved, the activity is put on a "ready" queue. The EU takes the first activity waiting on the ready queue, loads its context and executes it.



SSP: SIGNALING AND SCHEDULING PROCESSOR
EU: EXECUTION UNIT
PC: PERIPHERAL CONTROLLER

FIGURE 2 BLOCK DIAGRAM OF A NODE

3. ARCHITECTURE AND IMPLEMENTATION

A PN is a bus based multiprocessor as illustrated in Figure 2. It comprises a shared memory, a number of Execution Units (EU), a number of Peripheral Controllers (PC), and a Signaling and Scheduling Processor (SSP). The execution units, described later, are stripped down microcomputers. The peripheral controllers are similar to execution units except that they have special interfaces to I/O devices. The shared memory architecture allows efficient sharing of data structures and minimizes movement of data. In this architecture all communication is handled by the SSP.

Signaling and Scheduling Processor

The SSP provides for reliable transport of messages over channels that connect a PN to other processing nodes or devices. Additionally, it also provides for all the signaling and data movement between activities. As mentioned earlier we have separated execution of programs from the functions of scheduling and communication support. Thus, the execution units only execute programs under the direction of the SSP. An EU does not have to support a large local operating system. The peripheral controllers execute device specific programs and provide the necessary hardware interfaces. The SSP, on the other

hand, does not actually execute user programs but only moves signals around. So the requirements for the SSP are quite different from those of the execution units. In fact, the critical element of this architecture is the SSP and its interface to the execution units and the peripheral controllers.

Figure 3 illustrates the architecture of the SSP and its interface to the execution units and the peripheral controllers. The SSP consists of two major units: the Receiver and the Sender. The SSP signals the execution units through the "ready" queues. The execution units signal the SSP through the "wait" queues. The SSP also generates work for the peripheral controllers through the "output" queues. All data from the outside world comes through the peripheral controllers. The PC's signal the SSP via the "input" queues to request action upon the input data. All data movement between the execution units, the peripheral controllers and the SSP is done through the shared memory; only the signaling is done through the queues. A similar arrangement of shared queues was employed for interprocess communication in the SL10 packet switch [CLI76].

The division of functionality between the receiver and the sender in a SSP is as follows. The receiver implements the following functions:

— it monitors all incoming messages, external and internal to SSP, and processes them one at a time according to their urgency,

— for a request token it selects an activity that will provide the desired service, and

— it binds the activity to an EU by placing the activity object pointer in the ready queue of the EU.

The sender performs similar functions and these are:

— it monitors the wait queues of the EU and selects a synchronization object according to the urgency,

— it performs the function specified by the synchronization object.

— The function may be to send an external message or an internal synchronization signal.

— Upon completion of an external send operation, it puts a signal on the internal message queue of the receiver asking it to wake up the activity that had requested that send.

One key feature of this architecture is that all signaling is supported through hardware queues. The importance of implementing queues in hardware is two fold. First, the fifo queues provide an elastic buffer between the execution units and the SSP. Second, they provide a solution to the mutual exclusion problem which will result if the queues were to be implemented as a shared data structure in the common memory. This is because the reader and writer of a fifo have exclusive access to their ends of the device, hence, the need for any arbitration to access the fifo is eliminated. Furthermore, the usual hardware support for implementing mutual exclusion in multiprocessors are test-and-set operations. Although, test-and-set operations support well the implementation of controlled access to shared data, they are a poor mechanism for supporting asynchronous communication between activities. The reason is that test-and-set operations cannot affect scheduling. On the other hand, the hardware fifo's in our architecture not only provide an effective path to send signals but also affect the scheduling at the reader's end.

Another significant feature of this architecture is that it separates, in form of the SSP, the functions of communication and scheduling. This allows growth in the number of execution units and peripheral controllers, and reconfiguration without changing the communication structure.

The Receiver is very much like a sophisticated interrupt controller. Interrupts are asynchronous messages requesting scheduling of some activities. In the Receiver we have integrated that with other messages; hence all signals, hardware or software generated, are handled the same way. The Sender is also a scheduler except instead of scheduling execution units it schedules messages to be sent out and reschedules internal signals on the execution units through the Receiver.

The Receiver and the Sender are primarily data movers. However, the SSP needs only a limited instruction set. In order to achieve maximum throughput in the system, the SSP has to operate at a high enough rate such that no EU or peripheral controller is blocked waiting on the SSP. Then, the performance of the system is determined by the number of execution units and the peripheral controllers in the system. This presents an additional constraint on the number of execution units and peripheral controllers that can be efficiently used with a given implementation of the SSP. We intend to implement the SSP using high speed bit-slice processors, such as the AMD2900 series.

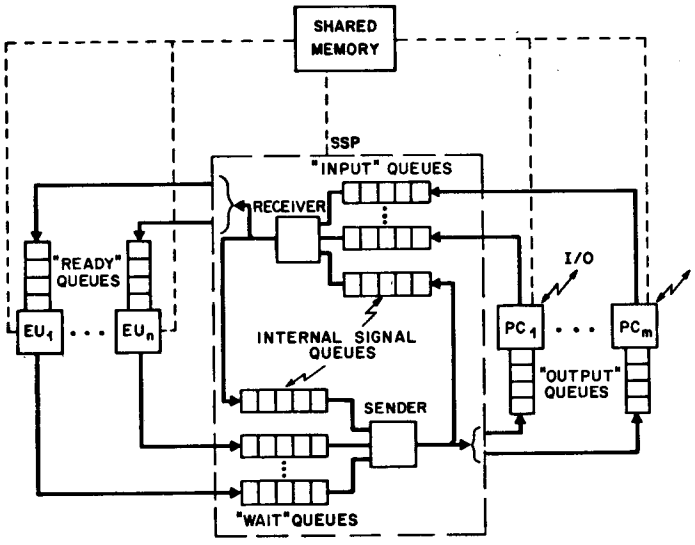


FIGURE 3 SIGNALING AND SCHEDULING PROCESSOR ARCHITECTURE

Hardware Scheduling of Activities

All computation in a PN is driven by the arrival of messages on channels much like the way computation proceeds in data flow machines. On the arrival of a message SSP examines the request token and schedules the desired activity to act upon the message. An activity typically computes sequentially and may then wish to communicate with some other activity. At that point it blocks, gives up the EU and is rescheduled only when the desired communication is accomplished. Whereas, it is an activity that voluntarily controls giving up of the EU it is executing on, it is the SSP that controls when an activity regains control of an EU. When an activity blocks due to a send operation, it will remain blocked until the specified message is transmitted to the other end of the channel and acknowledged to this effect. This acknowledgement is automatic and is not generated as a result of computation at the receiving end. For a receive operation, an activity will resume immediately if there is a message with the specified token already waiting to be read. If not, the activity will remain suspended until such a message arrives on the specified channel. The SSP essentially matches a token received on a channel to an activity corresponding to that token.

Execution Unit

Figure 4 illustrates the implementation of an EU. It is a Motorola 68000 microprocessor based single board computer. These boards have been enhanced to incorporate the 'Ready' queues. The 'Ready' queues form a part of the hardware signaling support. Each EU has memory that is locally addressable, called the local memory. The rest of the address space is shared by all the execution units and the SSP. The queues are implemented as hardware FIFO's similar to those proposed by Ward et.al. [WAR79]. A read corresponds to reading the front of the queue. A write corresponds to adding to the end of the queue. The ready queue also provides a Data Available signal that is monitored by a Queue Monitor. The function of a queue monitor is to arbitrate among the queues according to urgency and cause a switch in the CPU's thread of execution. We implement this function with a conventional interrupt controller such as INTEL 8259. It actually interrupts the MC68000 when a queue, with a priority higher than the activity currently running, becomes non-empty. There is now only one interrupt service routine that reads the interrupting queue to fetch an activity object pointer, restore the state of that activity and resume execution.

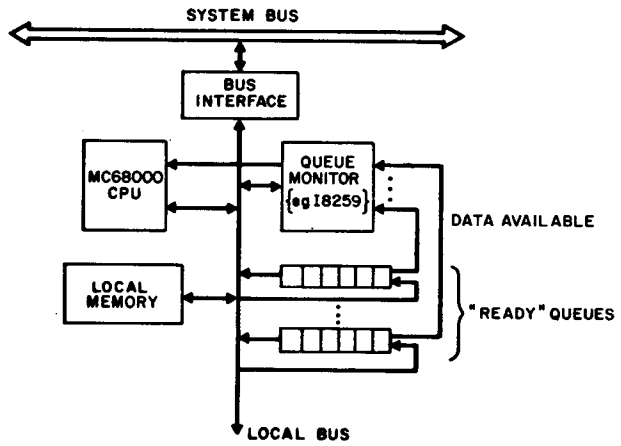


FIGURE 4 EXECUTION UNIT

4. AN EXAMPLE

We now illustrate the usefulness of this architecture with the implementation of a simple file server. The intent is to show how everything fits together by tracing the flow of data and signals through the various components of the system. We make no claims about the completeness of the file server model.

The file server supports open, close, read, write, seek and directory operations. Clients connect to the file server via channels and work with files using a predefined message protocol. The file server comprises a Storage Manager (SM) and Request Handlers (RH). Whenever a client connects to the file server a new instance of the Request Handler is created to serve requests from that client. A RH associated with a channel remains in existence until the client disconnects that channel. Another module, the Supervisor (SUP), authorizes the requests from clients for setting up new channels. We assume the existence of two peripheral controllers, a Disk Controller (DC) and a Network Interface Unit (NIU). Figure 5(a) depicts these modules and their relationship to the SSP.

Connect

We will consider the scenario for a request from a client to connect to the file server. The connect message is received, under control of the network interface unit, directly in a shared memory buffer (See Figure 5(b)). The NIU signals the SSP of the reception. The SSP recognizing that the message is a request to set up a new channel wakes up the supervisor and hands it a pointer to the received message for further processing. The wakeup is caused by writing the pointer to SUP's context on the ready queue of an appropriate EU. Upon resuming execution, SUP decides whether to accept or refuse the request for channel setup. Assuming that it decides to accept, SUP creates a new activity, RH, to service all subsequent requests from the client process at the other end of the channel. The SUP composes a response message indicating acceptance of the setup request and signals the SSP to send this message over a specified channel. The SSP signals the NIU to transmit that message.

File Read

Once a connection between the file server and a client process is established, all subsequent requests on that channel are serviced by the associated RH. Figure 5(b) shows the flow of signals that takes place in processing a file read request. Notice that the received request is deposited directly into the shared memory by the NIU and is never recopied again; only signals moving through the hardware queues cause various activities to act upon that request. Similarly, the data from the disk is read into the shared memory once by the disk controller and is not moved again until it is transmitted over a channel by the NIU.

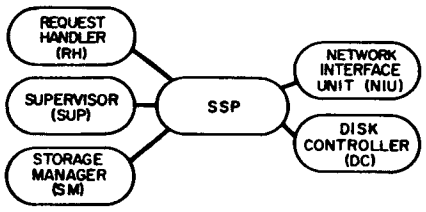


FIGURE 5 (a). ELEMENTS OF THE FILE SERVER

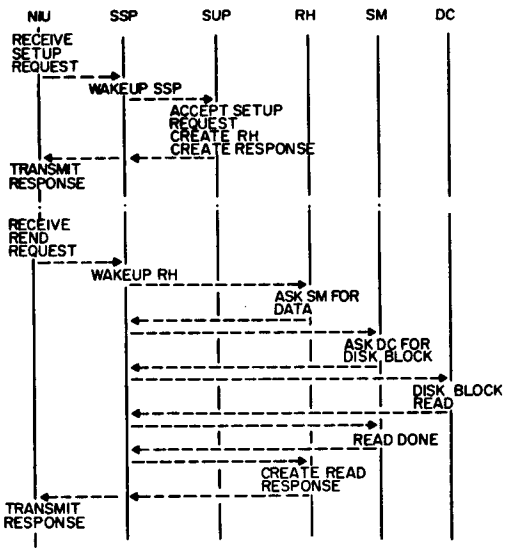


FIGURE 5(b) SCENARIOS FOR SETUP AND READ REQUESTS. NOTICE ALL SIGNALING TAKES PLACE THROUGH THE SSP.

5. PERFORMANCE

We expect several features of our architecture to contribute towards enhancing the performance of the system. The shared memory allows efficient data passing between units of the system by reducing the need for copying data. The hardware fifo's provide parallel paths for flow of signals between units. Each unit operates independently at its rate. The design of an EU is oriented towards the special function it is intended to perform. Thus, an EU only runs activities given their context pointers, the SSP sorts messages and schedules activities to operate on messages, and a PC performs device specific functions. In order to take advantage of all these features the application has to be partitioned appropriately. This architecture certainly cannot provide performance improvements for all classes of applications. We believe, however, that it lends itself well to a certain class of applications (such as transaction processing) that fit the model described in section 2. We are currently building a prototype system of four processors to investigate the system's performance for transactional applications. Further analysis needs to be done to examine the effects of interference in accessing the system bus and the shared memory. The effect of the SSP capacity on the performance and configuration of the system also remains to be analyzed.

6. CONCLUSION

We have described a multiprocessor system that incorporates into its architecture, at a very fundamental level, hardware support for communication and scheduling. The system treats synchronization and communication uniformly. It takes advantage of the intimate coupling between communication and scheduling and provides a special processor to support it.

An interesting picture emerges upon tracing the flow of information in Figure 3. The elements of the processing node are, in fact, organized in form of a circular pipeline with an inlet and an outlet to the outside world. Each component is coupled with its neighboring component via a fifo that serves both as a communication link and as an elastic buffer. Carefully exploited, such an arrangement can allow information to flow efficiently between components. Since all components in the system function independently, a high degree of concurrency is possible during its operation.

7. ACKNOWLEDGEMENTS

The authors are grateful to C. S. Roberts for his encouragement and suggestions during the course of this work.

8. REFERENCES

[ARV81] Arvind, and V. Kathail, "A Multiple Processor Data Flow Machine that supports generalized procedures," 8th Annual Symposium on Computer Architecture, 1981, pp. 291-302.

[AST81] A. Asthana, C. S. Roberts and G. K. Swanson, "Design of a Communications Kernel for Loosely Coupled Multiprocessors," Proc. 19th Annual Allerton Conference on Communication, Control and Computing, 1981.

[CHE79] D. R. Cheriton, et.al., "Thoth: A Portable Real-Time Operating System," CACM 22, 2, Feb. 1979, pp. 105-115.

[DIJ68] E. J. Dijkstra, "Cooperating Sequential Processes," Programming Languages (F. Genuys, ed.), Academic Press, New York, 1968.

[DEN80] J. B. Dennis, "Data Flow Super Computers," Computer, 13, 12, Dec 1980, pp. 48-56.

[DIT81] D. R. Ditzel, "Reflections on the High-Level Language Symbol Computer System," Computer, 14, 7, July 1981, pp. 55-67.

[HOA78] C. A. R. Hoare, "Communicating Sequential Processes," CACM, 21, 8, Aug. 1978, pp. 666-677.

[RIC81] R. Rice, "The Chief Architect's Reflection on Symbol IIR," Computer, 14, 7, July 1981, pp. 41-55.

[SMI77] A. J. Smith, "Multiprocessor Memory Organization and Memory Interference," CACM, 20, 10, Oct. 1977.

[SWA77] R. J. Swan, S. H. Fuller and D. P. Siewiorek, "Cm* - a modular Multiprocessor," Proc. AFIPS Press, Arlington, Va. 1977, pp. 637-644.

[WAR79] S. Ward, C. Terman, J. Sieber, and R. McLellan, "NU: The LCS Advanced Node," MIT report, 1979.

[CLI76] W. W. Clipsham, F. E. Glave and M. L. Narraway, "Datapac Network Overview," Proc. ICCC, August 1976, pp. 131-136.