Application Controlled IPC Synchrony – An Event Driven Multithreaded Approach

Susmit Bagchi and Mads Nygaard

Department of Computer and Information Science,
Norwegian University of Science and Technology (NTNU),
N-7491, Trondheim, Norway
{susmit, mads.nygaard}@idi.ntnu.no

Abstract. Interprocess communication (IPC) is an important phenomenon in distributed computing and operating systems. Microkernels of modern operating systems use synchronous IPC semantics for every individual process. On the other hand, a process may exploit non-blocking IPC semantics. In either case, the controlling mechanism belies in the hand of the underlying operating system. IPC monitors open up for misinterpretation of IPC timeout events due to thread unavailability in dynamic multithreaded systems. In this paper we propose a software architecture applicable to distributed systems, which confers the decision on IPC semantics during execution to the processes so that they can admix blocking and non-blocking semantics in a flexible way, case by case, as needed. Moreover, the concept of thread pool is introduced to eliminate the possibility of misinterpretation of IPC timeout events by monitors. Worker threads in a thread pool are effectively scheduled to minimize the waste of processing time and dynamic thread overhead. Event driven and multithreaded system models are diagonally opposite during execution. However, our architecture utilizes the benefits of an event driven model with that of a multithreaded model in a fruitful manner to exploit concurrency and protection. The software implementation of our proposed architecture is made as a middleware extension on the communication subsystem of Windows operating systems.

1 Introduction

Interprocess communication (IPC) is one of the most common concepts used in operating systems (OS) and distributed computing systems. It deals with how multiple processes can communicate among each other. In a distributed system, cooperating processes communicate by sending messages. High performance communication is a very critical facility in the distributed computing systems [3]. On the other hand, IPC monitoring enables the examination of any IPC between the source and destination machines [4]. Monitoring of IPC is useful for debugging, logging and security purposes. However, the performance of IPC and monitors is heavily dependent on the IPC semantics and protocols used to implement them [3]. Unfortunately, the IPC mechanism has been considered an expensive operation in the recent past [5]. However, "extensibility" has become an important phenomenon in OS research [5],

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2658, pp. 1012–1021, 2003.

and new IPC mechanisms along with monitors are being implemented as extensions of the basic kernel. There exists two main semantics in implementing IPC; i.e. synchronous and asynchronous. Such synchrony features are solely controlled by the underlying operating system [2]. In modern microkernels, synchronous IPC semantics is used [4]. This indicates that the communicating process gets blocked until the message is delivered to the destination or an error occurs. This method often leads to an unnecessary waste of execution time. In addition, every IPC message is stamped with a source specified timeout value. Unfortunately, a monitor may disrupt the interpretation of such timeout values if it does not have a thread ready to send or receive the message [4]. Eventually this leads to a loss of transparency of the IPC mechanism.

In this paper, we investigate a different scheme for IPC semantics and its realization, which encompasses a new control mechanism of IPC synchrony and concurrency. In effectively protected environments, we can provide flexible message synchronization relying on decisions taken by processes engaged in IPC. We provide a flexible and reliable mechanism so that individual processes may decide whether they should block for a message delivery and reception, or continue while processing the IPC as a background task. In addition, IPC concurrency is achieved in a better way using the event driven programming model by reducing opportunities for race conditions and deadlocks [1]. We establish an event driven and multithreaded architecture to exploit the benefits of both system models. The projected software architecture achieves the following set of benefits:

- **Design flexibility:** Our architecture assigns the synchronization decision authority to the processes themselves. Hence, they are allowed to generate individual jobs using different synchronization semantics and will be notified on the outcome of IPC events job by job. This adds a good amount of design flexibility to software applications.

- **Unwanted blocking time elimination:** Due to admixing blocking and non-blocking IPC events at the application level, the applications can save execution time by eliminating unwanted waiting for message delivery and response in some specific cases.

- **Elimination of misinterpreted timeout by monitors:** The proposed architecture uses a thread pool concept that eliminates the overhead of creating numerous dynamic threads while handling IPC requests in a multiprocessing environment. Hence, our software architecture eliminates the possibility of misinterpreting timeout events due to message delivery/response.

- **Combining benefits of event driven and multithreaded concurrency:** The designed software middleware treats the IPC requests as discrete events. Hence, processes get immediate attention on IPC events. An event driven approach makes it easier to ensure protection against race conditions and deadlocks [6]. On the other hand, due to our multithreaded model, the benefit of execution concurrency is achieved.

• **Efficient CPU utilization:** The threads are scheduled by the middleware, and this scheduling is based on IPC event generations or completions. Such a scheduling ensures that idle threads will never spend unnecessary processor time cycles.

The remainder of our paper is composed as follows. In section 2, we define different IPC semantics and different programming models. In section 3, we construct a model for application controlled synchrony. This establishes a software architecture combining the multithreaded and event driven IPC paradigms and introduces the thread pool concept. In section 4, we describe our implementation and related data structures. In section 5, we conclude the paper.

2 IPC Semantics and Programming Models

A good interprocess communication facility is central to any distributed system and provides access to resources distributed over a network in a uniform manner. Generally, the primitives used for IPC are *message passing*, *remote procedure call* and *transaction communication*. In the case of message passing, a set of messages is communicated among coordinating processes. At the lowest level, message passing is the only means of communication in distributed systems [9]. But communication transparency is desirable by providing a higher level of abstraction. The remote procedure call (RPC) concept may be used to achieve this goal [9]. The RPC mechanism is built on top of a client/server model and is widely used. Finally, service oriented request/reply communication and multicast may be combined to form a third level, transaction communication. The transaction communication depicts a set of asynchronous request/reply messages without sequential constraints like transactions in a database [9]. All these primitives are mainly based on two different types of semantics; i.e. synchronous and asynchronous communication.

2.1 IPC Semantics and Monitors

The presence of monitors complicates the exploitation of synchronous/asynchronous communication. Three possible cases may arise [4]: (1) IPC monitors may change the identity of source and destination, (2) Monitors may wish to hide the timing of the destination's receipt of the request from a source, and (3) Monitors themselves may be of different trust levels. Hence, desirable aspects associated with IPC are reliability, error handling and timeout expiry handling. In addition, IPC monitors may redirect the IPC request in some cases [8].

2.2 Programming Models

An event driven programming model simplifies concurrency issues by reducing the risks for race conditions and deadlocks [6]. On the other hand, a multithreaded programming model ensures readability and maintainability of code [1]. The sources of conflation for task management and stack management are identified in [1]. The

event driven programming model suggests that the gain in concurrency cannot be achieved without manual stack management, which is difficult to attain. However, with these two diagonally opposite programming paradigms, one can cautiously combine the advantages of both models [1].

3 Synchrony, Events, and Worker Threads

Usually, the issues related to IPC synchronization are controlled by the operating system, while the processes use this as a service. Modern OS microkernels implement synchronous IPC semantics [4]. But in some cases, allowing processes to decide about synchronization adds a lot of operating flexibility. In addition, it saves unnecessary blocking time in particular cases and helps a process to admix blocking and non-blocking semantics, as it needs during execution. Moreover, it is possible to combine the advantages of an event driven architecture with that of a multithreaded model to increase execution concurrency of multiple IPC requests on a timeshared basis. We use the thread pool concept to eliminate the overhead of dynamic thread creation and related timeout events.

3.1 Application Controlled Synchrony Model

Modern microkernels use synchronous IPC semantics where a source is blocked until the destination is ready or an error occurs. We have designed a modified architecture where the application or source process can decide, any time during execution, whether to get blocked until the destination process receives the message, or to proceed executing after the message is queued in the system. The calling process decides this while invoking an IPC event. Hence, the process may explicitly control IPC semantics. The task of the operating system will be to provide a uniform interface for message delivery and to report to the calling process about the result. The result may be success or failure. But the message synchrony decision will belong to the calling task.

3.2 Event Driven Architecture

As event driven and multithreaded concurrency models are diagonally opposite, we have tried to combine the advantages of both models. A compromising approach is cooperative threads management. The proposed software architecture is based on an event driven system model. The new approach to achieve cooperative thread management is to organize a system as a collection of event handlers. Hence, sending or receiving a message are two distinct events, and are attached to corresponding event handlers. Every process will have two distinct event handlers to get the result notification from its worker threads. Hence, we consider the “SendRequest” and “WaitResponse” operations as two distinct operations invoked by any process. Moreover, this will allow the IPC subsystem to process multiple blocking and non-blocking IPC events concurrently exploiting event driven multithreading. Our

architectural model employs a three-tire middleware extension of a communication subsystem. The three different levels are as follows: level 1: the processes; level 2: the IPC subsystem; level 3: the thread pool. Hence, there exists two interfaces i.e., process to IPC subsystem interface (level 1-level 2) and IPC subsystem to thread pool interface (level 2-level 3). The interaction among the three levels is shown in Figure 1.

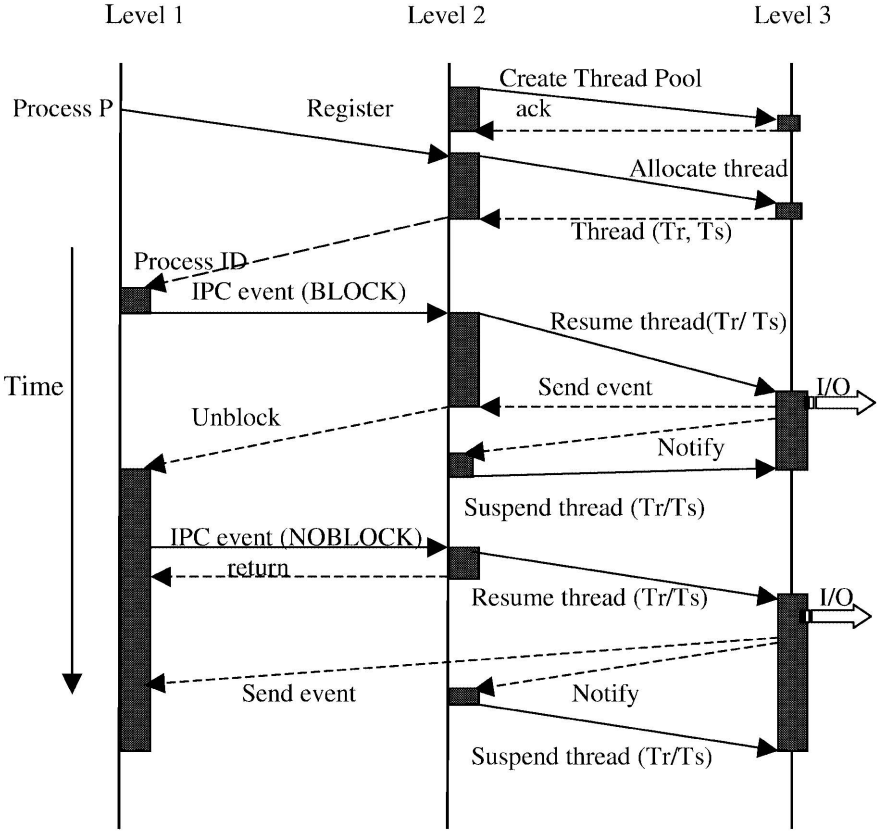

Fig. 1. Three-tire interaction model

During system initialization, level 3 creates a pool of suspended threads, and the thread stacks are initialized. After receiving confirmation from level 3 about successful creation of system resources (thread pool and thread stacks), level 2 creates its own data structures and becomes ready to handle IPC events. As shown in Figure 1, a process must register before requesting any IPC operation, and the process should nominate two event handlers. On successful registration, unique Process IDs will be returned to the individual processes for future use, and two threads will be allocated

from the pool to each registered process. Registered processes belong to level 1 and may generate an IPC event explicitly indicating the type of event (BLOCK or NOBLOCK) as shown in Figure 1. Depending on the event types, level 2 schedules the threads to accomplish the requested service. On completion, the thread can notify the result either to level 2 or to level 1, depending on the type of IPC event. Thread allocation belongs to level 3, but scheduling allocated threads in various phases, and its status maintenance, belongs to the level 2. This event driven design ensures immediate thread scheduling upon the arrival of an IPC request, and prevents idle threads from gaining CPU resources. When a thread notifies level 2 about the end of a requested service, the scheduler in level 2 changes the thread state to suspended and marks it as idle. Hence, no idle thread uses any excess CPU cycles. In addition, our event based system model reduces the occurrence of race conditions and deadlocks. Detailed descriptions of the necessary data structures and algorithms are given in Section 4.

3.3 Concurrent Thread Pool

Our concurrent thread pool is a pool of threads in suspended state. For each send and receive operation, allocated working threads from the thread pool are responsible for completing the operations concurrently. Every thread in the pool has an individual thread stack. The thread pool is considered as a system resource containing suspended threads. When a process registers in the IPC subsystem, two working threads are allocated (by level 3) and the thread stacks get initialized. In addition, two job queues (one “JobQ” for “SendRequest” method and another “JobQ” for “WaitResponse” method) are allocated to each registered process (by level 2). In this model, a thread Tr acts on the receive “JobQ” of a process P and the other thread Ts acts on the send “JobQ” of the process P. When process P exits, Tr and Ts are returned to the thread pool in a suspended state. Our multithreaded event driven IPC system is capable of handling multiple requests concurrently for all processes without contention, and frees the system from the burden of dynamic thread creation, which may lead to misinterpreted timeouts in IPC monitors [4]. Additionally, allocated threads are efficiently scheduled to minimize consumption of unnecessary CPU resources.

4 Implementation Issues

We have implemented this application controlled synchrony in our event driven multithreaded execution model as a middleware on top of a sockets based communication subsystem of Windows OS. The data structures, interface definitions and algorithms of our experimental middleware are described below. The overall software architecture is presented in Figure 2.

4.1 Data Structures

The important data structures are described below.

• **Job queue (JobQ):** The JobQ data structure is used to queue the job requests for send or receive generated by processes. Every process has two distinct queues of

finite length, one for "SendRequest" and one for "WaitResponse". The main members of JobQ are "JobNo", "Status" and "Type". "JobNo" represents the job sequence number assigned by the calling process. While "Status" and "Type" indicate the status of the job (serviced or pending) and nature of call (blocking or non-blocking as requested by the calling process) respectively.

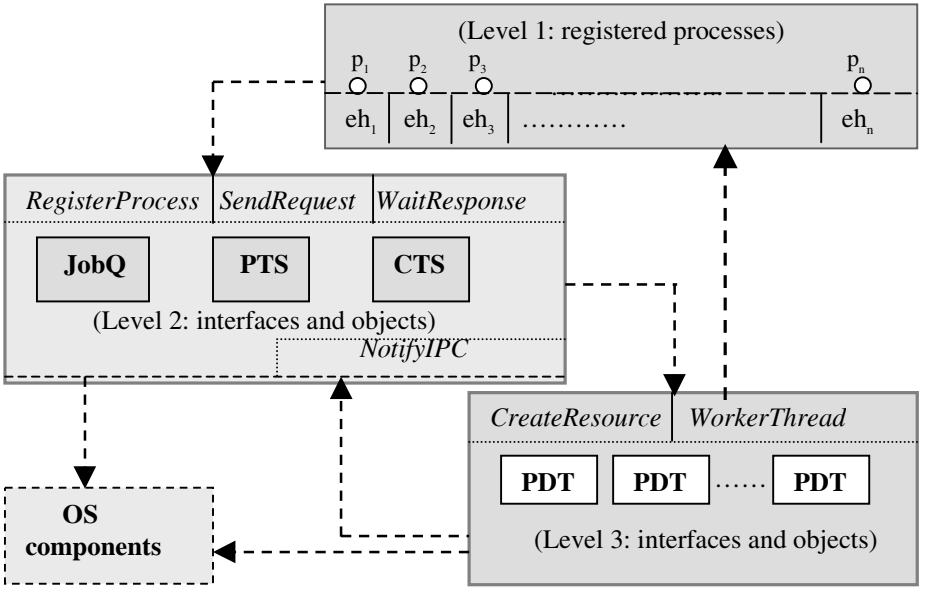

Fig. 2. Three-tire software architecture

• **Private data for thread (PDT):** Every working thread in the thread pool contains this on their individual thread stacks. One of the main members is "ProcessStackNo", representing the index in the process table. This field is used to get the entry points of the event handlers (eh) of the calling process. The member "pThreadJobQueue" holds a reference to the job queue (send or receive) on which the thread should act. Further, the member "ThreadIndex" gives the index for the associated thread. Finally, the member "pThreadStateInfo" holds the entry point in the IPC subsystem for this thread.

• **Common thread structure (CTS):** This data structure is used to register the context of every thread. The main members of this structure are "ThreadHandle" representing a handle to the thread and "ThreadState" representing the execution mode of the thread at any point of time. While the member "ThreadIntifier" is used to store the identifier assigned by the underlying OS and is used to access the thread from level 2 in the middleware.

• **Process table (PTS):** The execution context of every registered process is stored in this object. Among its members are "`ProcessID`", "`ProcessEventHandler`", "`pThreadAlloc`" and two JobQs. The member "`pSendJobQueue`" represents the job queue used by the "`SendRequest`" method and "`pRecvJobQueue`" represents the same for the "`WaitResponse`" method. The field "`ProcessID`" holds the distinct process ID assigned to any registered process. The "`ProcessEventHandler`" holds the event handler entry points for processes to notify the result of an IPC asynchronously. The member "`pThreadAlloc`" holds a reference to the global thread data of the threads attached to this process.

4.2 External Interface Definitions

The interface offered by our experimental middleware is given below.

• **`RegisterProcess()`:** This function is used to register a process in the IPC subsystem. It takes the entry points of process event handlers as the arguments and returns a unique "`ProcessID`".

• **`SendRequest()`:** This function is used by a process to send a message to another process. The "`Option`" argument of the function shall be "BLOCK" or "NOBLOCK" to indicate that it is a blocking or non-blocking call.

• **`WaitResponse()`:** This function is used to raise an exception, i.e. IPC event, by a registered process in order to receive/wait for a response message. The "`Option`" argument of the function shall be "BLOCK" or "NOBLOCK" to indicate that it is a blocking or non-blocking call.

4.3 Internal Interface Definitions

The main interlayer functional components are:

• **`CreateResource()`:** This function is responsible for resource allocation and initialization. It belongs to level 3. It returns "Success" or "Failure".

• **`WorkerThread()`:** This function describes the worker thread procedure. It belongs to level 3. It returns "Null".

• **`NotifyIPC()`:** This function is used by threads to notify level 2 about the end of a requested IPC service. In level 2, the thread scheduler gets invoked, and the thread is transferred into the suspended state and marked as idle.

4.4 Algorithms

The main algorithmic parts to accomplish our IPC system model are depicted in the following procedural steps.

```
//System initialization
Proc SysInit:
  begin
    Create_Process_Table;
    Create_Thread_Pool;
    Init_Thread_Stack;
    Regd._Callback_Level2;
    Suspend_All_Threads;
  end begin
```

```
//Process registration
Proc RegisterProcess:
  begin
    Update_Process_Table;
    Create_Job_Queues;
    Allocate_Thread;
    Init_Thread_Stack_Data;
  end begin
```

```
//Worker thread functions
Proc WorkerThread:
  begin
    Check_Private_Data;
    do
      Pick_From_Job_Queue;
      Invoke_IO;
        if(!Blocking)
          Send_Evt_Process;
        end if
      Clean_Up_Memory;
      Send_Evt_To_Level2;
    end do
  end begin
```

```
//SendRequest or
//WaitResponse events
Proc Send/Wait:
  begin
    Enqueue_Job;
    Activate_Thread;
      if(Blocking)
        Wait_Thread_Event;
      else
        Return_Result;
      end if
  end begin
```

5 Conclusions and Future Work

Our application controlled synchrony and event driven multithreaded IPC model enable concurrent message handling. As a pool of threads is created in the system beforehand, threads may be allocated to the processes on demand. This eliminates the overhead of creating threads dynamically while handling IPC requests. Hence, our proposed architecture eliminates the possibility for misinterpretation of timeout events due to non-availability of threads. In addition, our software architecture enables the processes to admix blocking and non-blocking IPC semantics in a flexible manner as the decision authority belongs to the processes. By admixing blocking and non-blocking IPC semantics, the processes save execution time by eliminating unwanted waiting for message delivery in some specific cases. Due to the event driven behaviour of the proposed model, immediate attention to IPC requests is achieved. In this software architecture, we have tried to combine the advantages of event driven systems and the multithreaded programming model. The event driven approach makes it easier to reduce the possibilities for race conditions and deadlocks. On the other hand, our architecture achieves the benefit of concurrent execution through multithreading. However, the thread scheduling mechanism relies on the middleware (level 2), and such scheduling is done on the basis of an IPC request or the completion notification of an IPC event from a worker thread. In this way we ensure that idle threads will never employ unnecessary processor cycles. When a process expires or exits, the workers threads previously allocated to this process returns to the thread pool.

The challenge of our execution model is that suspended threads preoccupy resources such as memory. Although such a waste is not dominant, processes may not use the entire thread population at any time. So, there may always be a few idle threads in the pool. On the other hand, if the system contains numerous processes, then the total thread population in pool may not be large enough to serve all the processes. We are currently investigating modifications to our present implementation to eliminate such difficulties.

References

1. Adya, A., Jon, H., Marvin, T., William, J. B., John, R. D., June 2002. Cooperative Task Management without Manual Stack Management, In the proceedings of the USENIX Annual Technical Conference, CA.
2. Colouris, G., Jean, D., Tim, K., 2002. Distributed Systems Concepts and Design, Addison-Wesley, Third edition, pp. 125–158.
3. Goscinski, A., 1991. Distributed Operating System, Addison-Wesley Publishing Company, pp. 204–207, 871–874.
4. Jaeger, T., Jonathan, E. T., Gefflaut, A., Park, Y., Kevin, J. E., Jochen, L., September 2002. Synchronous IPC Over Transparent Monitor, In 9th SIGOPS European Workshop, Denmark.
5. Jochen, L., Kevin, E., Sebastian, S., Hermann, H., Gernot, H., Nayeem, I., Trent, J., May 5–6, 1997. Achieved IPC Performance, In 6th Workshop on Hot Topics in Operating Systems (HotOS), Massachusetts.
6. John, O., January 1996. Why threads are a bad idea (for most purposes), In USENIX Technical Conference (Invited Talk), Austin, TX.
7. Kent, B. K., Muzio, C. J., Shoja, C. G., April 2001. Remote Transparent Execution of Java Threads, In proceedings of the HPC-2001, Seatle, pp. 184–191.
8. Liedtke, J., Clans & Chiefs, March 1992. In Architektur von Rechensystemen, Springer-Verlag, In English.
9. Randy, C., Theodore, J., 1997. Distributed Operating Systems & Algorithms, Addison-Wesley, pp. 48–49, 123–124.