Process Synchronization and IPC

CRAIG E. WILLS

Worcester Polytechnic Institute ⟨*cew@cs.wpi.edu*⟩

Process synchronization (also referred to as process coordination) is a fundamental problem in operating system design and implementation whenever two or more processes must coordinate their activities based upon a condition. A specific problem of synchronization is mutual exclusion, which requires that two or more concurrent activities do not simultaneously access a shared resource. This resource may be shared data among a set of processes where the instructions that access these shared data form a critical region (also referred to as a critical section).

Processes involved in synchronization become indirectly aware of each other by waiting on a condition that is set by another process. Processes can also communicate directly with each other through interprocess communication (IPC). IPC causes communication to be sent between two or more processes. A common form of IPC is message passing.

UNDERLYING PRINCIPLES

Process synchronization and IPC arose from the need to coordinate concurrent activities in a multiprogrammed operating system. A fundamental synchronization problem is mutual exclusion, which was described by Dijkstra [1965]. Two other fundamental synchronization problems are the producer/consumer problem, where one process produces data to be consumed by another process, and the readers/writers problem, which occurs when multiple readers and writers want access to a shared object such as a database.

Many approaches are available for solving mutual exclusion and synchronization problems, but there are a number of issues concerning the implementation of solutions. The primary issues are whether the solution requires processor synchronicity (uninterruptible processors), which works only on a uniprocessor, versus store synchronicity (atomic memory references); whether the solution requires busy waiting, the continued polling of a condition variable; whether the mechanism has inherent problems with programmer error; whether the synchronization solution leads to starvation, where a process is indefinitely denied access to a resource while other processes are granted access to the resource; and whether the solution can deadlock, where a set of processes using shared resources or communicating with each other are permanently blocked. The classic dining philosophers problem is often used to test synchronization solutions because it has the potential of leading to both deadlock and starvation.

Interprocess communication problems generally involve direct communication between two or more processes, in contrast to synchronization, where processes communicate indirectly by waiting on or setting a condition. IPC mechanisms generally communicate by passing messages between processes. There are a number of issues: whether communication is addressed directly to a process or through an intermediate mechanism such as a mailbox; whether the message-passing mechanism allows messages to be buffered if the receiving process is currently not ready to receive a message; whether the send operation blocks if there is no space to buffer a message; whether the receive operation blocks if there is no message available; whether messages are fixed or variable size; and whether the mechanism uses synchronous reception, where messages are received only when the receive operation is invoked, or an asynchronous approach where message handlers are used.

BEST PRACTICES

This section discusses best, or common, practices for synchronization and IPC. More concurrent programming examples can be found in Ben-Ari [1990], Raynal [1986], and Brinch Hansen [1973]. Andrews and Schneider [1983] provide a survey of synchronization and IPC techniques.

Synchronization Mechanisms. The mechanisms for synchronization are divided into four types based on their level of implementation and support: software only, hardware support, operating-system support, and language support. In addition, hybrid solutions exist that combine more than one approach.

Software-based synchronization solutions use shared variables to control access to a critical region. Dekker was the first to devise a software solution that correctly handles the mutual exclusion problem among a set of processes [Dijkstra 1965]. Peterson [1981] provided a simpler solution of the same problem.

One of the simplest ways to enforce mutual exclusion is to disable hardware interrupts at the start of the critical region, thus ensuring that the process does not give up the CPU (through a context switch) before completing the critical region. Another hardware-based approach is to use special instructions to implement mutual exclusion. One such instruction is `Test_and_Set`, which returns the previous value of a target variable and sets the target to the given value. This instruction is performed in an atomic manner so a context switch cannot occur in the middle of it. The advantages of this machine-instruction approach are its simplicity and the fact that it works for any number of processors and processes. Its primary disadvantage, particularly on a uniprocessor, is its use of busy waiting.

To avoid problems with busy waiting, semaphores, an important synchronization primitive, can be constructed by adding process-coordination support to the operating system. The concept of a semaphore was first introduced by Dijkstra [1968]. Semaphores are data structures consisting of an identifier, a counter, and a queue; processes waiting on a semaphore are blocked and placed on the queue; processes signaling a semaphore may unblock and remove a process from the queue; and the counter maintains a count of waiting processes. Not only can semaphores be used for mutual exclusion, but they also provide a mechanism to solve other synchronization problems.

Some programming languages provide constructs to implicitly guarantee mutual exclusion. One such construct is a monitor [Hoare 1974], which permits only one process to be executing in a monitor at a time. Many other synchronization primitives have been proposed, but in general can be expressed in terms of the solutions already given. Some of these primitives are critical regions, serializers, path expressions, and event counts and sequencers.

Modern operating systems have migrated from monolithic systems written for a uniprocessor in which disabling interrupts were used to access shared data structures. In modern systems, complex locks, which combine the use of spin locks with the semantics of semaphores, are used for better correctness and performance on uni- and multiprocessors.

IPC Mechanisms. The simplest form of message passing is to send messages directly from one process to another.

Typically a process can buffer one message so that both the send and receive operations may potentially block. Another direct message-passing mechanism is implemented with rendezvous so that both operations block until the receiving process has actually copied the message from the sender.

Rather than send directly to process, a more common approach is to define another operating-system abstraction called a mailbox (also referred to as a port). Mailboxes are buffers that hold messages sent by one process to be received by another process. Thus there is indirect communication between the two processes.

A special case of IPC is the pipe abstraction, which is a unidirectional, stream communication abstraction. Another IPC form is software interrupts, which associate interrupts sent to a process with interrupt handler routines.

SUMMARY

In summary, synchronization and IPC are fundamental to multiprogrammed operating system design. Primitives that solve problems such as mutual exclusion and producer/consumer range from software-only approaches, to special hardware instructions, to primitives constructed by the operating system and programming languages. The adoption of traditional synchronization primitives for multithreaded, message-based operating systems running in a multiprocessor environment is leading to work on hybrid approaches for synchronization.

REFERENCES

ANDREWS, G. R. AND SCHNEIDER, F. B. 1983. Concepts and notations for concurrent programming. *ACM Comput. Surv. 15,* 1, (March) 3–43.

BEN-ARI, M. 1990. *Principles of Concurrent and Distributed Programming.* Prentice Hall, Englewood Cliffs, NJ.

BRINCH HANSEN, P. 1973. *Operating Systems Principles.* Prentice Hall, Englewood Cliffs, NJ.

DIJKSTRA, E. W. 1965. Solution of a problem in concurrent programming control. *Commun. ACM 8,* 9 (Sept.), 569.

DIJKSTRA, E. W. 1968. *Co-operating sequential processes. In Programming Languages,* F. Genuys, Ed., Academic Press New York, 43–112. Reprint of Tech. Rep. EWD-123, Technological Univ., Eindhoven, the Netherlands (1965).

HOARE, C. A. R. 1975. Monitors: An operating system structuring concept. *Commun. ACM 17,* 10 (Oct.), 549–557. Erratum in *Commun. ACM 18,* 2 (Feb. 1975), 95.

PETERSON, G. L. 1981. Myths about the mutual exclusion problem. *Inf. Process. Lett. 12,* 3 (June) 115–116.

RAYNAL, M. 1986. *Algorithms for Mutual Exclusion.* Wiley, New York.