

SEMAPHORES

A semaphore is a higher level mechanism for controlling concurrent access to a shared resources.

Instead of using operations to read and write a shared variable, a semaphore encapsulates the necessary shared data, and allows access only by a restricted set of operations.

Because a semaphore has the power to suspend and wake processes, semaphores and similar higher-level constructs require the co-operation of the operating system and/or programming language run-time system.

There are two operations on a semaphore S. Worker processes can wait() or signal() a semaphore. For historical reasons, the wait and signal operations are sometimes abbreviated as P and V respectively.

Note that with semaphores, worker processes do not execute a potentially wasteful busy-waiting loop.

SEMAPHORE OPERATIONS

Wait(): a process performs a wait operation to tell the semaphore that it wants exclusive access to the shared resource.

If the semaphore is empty, then the semaphore enters the full state and allows the process to continue its execution immediately.

If the semaphore is full, then the semaphore suspends the process (and remembers that the process is suspended).

Signal(): a process performs a signal operation to inform the semaphore that it is finished using the shared resource.

If there are processes suspended on the semaphore, the semaphore wakes one of the up.

If there are no processes suspended on the semaphore, the semaphore goes into the empty state.

TYPES OF SEMAPHORES

The semaphore we have described is a binary semaphore. It allows only one process at a time to access the shared resource. The textbook describes a more general semaphore that allows $N > 1$ processes to access the resource simultaneously. Instead of having states empty and full, it uses a counter S . The counter is initialised to N , with $S = 0$ corresponding to the full state.

Note that different implementations of semaphores use different policies for choosing which process to wake during a signal action. A blocked set semaphore maintains a set of waiting processes and wakes one at random; while a blocked queue semaphore wakes processes in the order in which they were suspended.

XCircular SEMAPHORE MODEL

```
// Semaphore

Process Sem, Empty, Full, Wake, FullWaiting

Empty      <- pA goA Full + pB goB Full +
             vA Empty + vB Empty
Full       <- pA sleepA FullWaiting + pB sleepB
FullWaiting +
             vA Empty + vB Empty
FullWaiting <- vA Wake + vB Wake
Wake       <- wakeA Full + wakeB Full

Sem      <- Empty
```

OTHER CONCURRENCY CONTROL MECHANISMS

In comparison to the shared variable approaches we studied earlier, semaphores are a more structured, higher-level concurrency control construct. Worker processes no longer have to manage the details of the mutual exclusion solution, they just call `wait()` and `signal()`.

Programming with semaphores still requires locating and protecting each location where mutual exclusion is needed. Forgetting a `wait()` or `signal()`, or putting them in the wrong place can cause difficult-to-diagnose problems.

We will briefly mention two other concurrency control mechanism made available in various programming languages: *monitors* and *transactions*.

MONITORS

Monitors can be thought of as an object-oriented extension of the idea of a semaphore.

A monitor *encapsulates* a set of shared variables or data and a set of operations for manipulating the data, *and* it ensures mutual exclusion for each of the operations.

Worker processes are thus freed from the responsibility for ensuring mutual exclusion. Provided the encapsulation is adhered to, it becomes impossible to interact with the shared data without correct mutual exclusion, eliminating many possibilities for error.

In Java for instance, every object is potentially a monitor. Monitor operations are specified by putting them in a method and marking the method with the `synchronized` keyword.

TRANSACTIONS

Databases that provide concurrent access to multiple processes face the same data consistency issues as concurrent programs. A complex database update may require multiple queries and multiple modifications to the database; if the data being updated is modified by another process during the operation, the result could be incorrect.

A *transaction* is a set of read and write operations on a database that expects to see the database in a consistent state, and leaves the database in a consistent state after it has completed.

In database applications, it is often the case that there is no fixed set of transactions that can be identified ahead of time. A database may be accessed by different programs, and new transactions may be added over time. Because there is not fixed set of operations, the data cannot be completely encapsulated in a monitor.

Concurrency control is provided to transaction programmers in the form *read* and *write locks*, which must be obtained before reading or updating a data item, respectively.

A read lock prevents other processes from obtaining a write lock; while a write lock prevents other processes from obtaining both read and write locks.

TRANSACTION LOCKING SCHEMES

There are various locking schemes that differ according to the order in which locks can be made and released during a transaction. The different locking schemes have different properties in terms of the possibility of deadlocks and relative efficiency for different database access patterns.

There are also *lock-free* transactional schemes that rely on the ability of transactions to potentially fail and be repeated. In a lock-free system, the write operations of a transaction are saved locally until the end of the transaction. If the set of writes cannot be consistently carried out, the whole transaction is re-tried from the beginning.

Lock-free schemes have the advantage of not requiring the programmer to consider locking logic, and having lower run-time overhead – provided that contention only rarely occurs

Although originating in the database world, transactional systems are now being more widely used as a concurrency control solution. At the time of writing (March 2007), Intel for instance are researching hardware support for *transactional shared memory*.