Semaphores, Are They Really Like Traffic Signals?

Carlos M. Tobar, Juan M. Adán-Coello, Ricardo Luís de Freitas

Engenharia de Computação – Pontifícia Universidade Católica de Campinas
13.086-900 – Campinas – SP – Brazil

{tobar,juan,rfreitas}@puc-campinas.edu.br

Abstract. *Semaphores are software mechanisms designed to synchronize processes. The name seems to recall a traffic signal but diversely from the original idea behind them, different implementations can be found in popular Operating Systems as well as different definitions in the most adopted introductory books. Due to these differences, care is necessary in knowing if an implementation of semaphores is compatible with the definition found in the adopted book, otherwise could become confused to students. This article presents and compares semaphores definitions and implementations, rescuing three types of them. It is a warning to teachers on semaphore semantics and implementations, considering that they are usually used by applications programmers where kernel mechanisms cannot be used.*

1. Introduction

Semaphores are some of the mechanisms that can support *mutual exclusion* in order to avoid race conditions between processes. In doing so, they *synchronize* the processes regarding the access to a common resource.

There are several ways in which semaphores were designed and implemented since Dijkstra, in mid 1960s, applied the idea of mutual exclusion (THOCP, 2008). In a one-page paper he introduced the mutual exclusion situation along with the concepts of *critical section* and the *deadlock problem* (Apt, 2002).

Dijkstra's original idea on semaphores is discussed in his manuscript EWD 51, written in Dutch (Apt, 2002). After being implemented, it was presented in a second occasion in an American journal (Dijkstra, 1968). There he presents a data type with value domain restrained to integers, together with three possible operations that consider one semaphore instance: one for initialization, and the other two for conditional modification of the semaphore value and conditional state modification of one process that is operating or one that operated on the semaphore.

Each semaphore instance has an associated control structure for processes. If a process performs a *P* operation on a semaphore instance which has the value zero, that process has its execution *blocked* and is added to the semaphore structure, until a future execution of a *V* operation by another process. When that other process operates on the semaphore by performing a *V* operation, and there are processes in the respective structure, one of them is removed from there and goes to the *ready* state to resume execution. The semaphore value is not modified. It was, according to Dijkstra (1968), logically immaterial which of the processes is chosen in some contexts. But it might not happen that any *blocked* process stay in the semaphore structure indefinitely (Dijkstra,

1971). If there is no process in the semaphore structure its value is incremented by one.

The *P* and *V* operations must be atomic, *i.e*., the process whose *P* or *V* operation is under execution may not lose the CPU during that execution to run another operation, at least on the same semaphore. They are critical sections that require mutual exclusion.

Although semaphores are Dijkstra's idea and were considered effective synchronization mechanisms at the Operating System (OS) level, since their appearance: they were implemented differently in different OSs; they were the base of other mechanisms for process synchronization; their operations received different naming conventions throughout their implementations and theoretical definitions in books.

All these facts very likely cause confusion and misunderstanding to students who are assigned to learn process management as happens in OSs introductory courses.

Considering the importance of learning and using synchronization mechanisms by application programmers where kernel mechanisms cannot be used, and without considering symmetric multiprocessing or *multicore* computer architectures, in the following section are discussed different versions for definitions of *P* and *V* operations, or equivalents, which are present in the most highly considered introduction books on OSs. Implementations in the most popular OSs are also presented for semaphores and other mechanisms based on Dijkstra's idea. The goal is to show that some of the implementations do not offer exactly what must be expected and that implementations are not compatible to definitions.

The paper context, regarding characteristics of hardware and basic software, is related to computers with one *mono core* CPU and *multitasking* scheduling, although its information is valuable to other scenarios.

2. Semaphore Definitions

Basically there are three different types of semaphore definitions in the most adopted OSs books. They are discussed next.

First Definition – "Weak" Semaphores

Silberschatz et al. (2004) and Stallings (2004) discuss semaphores quite different from Dijkstra. First is the naming convention, with the substitution of *P* by *wait* and *V* by *signal*. Second, the semaphore value is always decremented in the equivalent *P* implementation and incremented in the *V* equivalent implementation. For Dijkstra, zero is the minor value for a semaphore. With this definition, the semaphore value could be negative while there are blocked processes waiting for their execution to continue. There is also the necessity to re-execute the *P* equivalent implementation by the process that is removed from the semaphore structure, which is a queue. This re-execution is implemented by not modifying the PC register value when the process goes blocked.

This semaphore semantics allows starvation to happen because there is nothing that prevents new processes from performing an equivalent P operation that decrements the semaphore value before the recently removed process has access to the CPU and restarts its execution, by re-executing its equivalent P operation, which results in a new insertion in the structure. According to Reek (2002), this is a weak semaphore, except by the non busy-waiting implementation of the *P* equivalent operation, which he considers part of a

weak semaphore and is not present in this first definition.

Second Definition – Blocking-Set Semaphores

Tanenbaum (2008) and Tanembaum and Woodhull (2006) present semaphores as data types with three operations. Their implementation is almost similar to Dijkstra´s definitions, except by the naming convention that substitutes *P* by *down* and *V* by *up*, and by the policy used to release one process when a *V* equivalent operation is executed. The semaphore value is non-negative. An *up* on a semaphore, with processes on its structure, causes a random pick of what process is removed, differently from Dijkstra's ideas (Dijkstra, 1971). After an *up*, the semaphore will still be with a zero value, but there will be one fewer process waiting on it.

Because all computational random functions are really pseudo random generators, it is possible, although unlikely, that a process is never removed from the semaphore structure, while the value is zero, configuring a starvation situation. This same problem is considered in Dijkstra´s definitions by some authors like Reek (2002), who sees no guarantee of justice in the picking strategy. This definition is named blocking-set by Stark (1982), because the group of blocked processes may be modeled as a set, from which a random process is unblocked when an equivalent *V* operation is executed.

Third Definition – Blocking-Queue Semaphores

Deitel et al. (2004) and Deitel (1984) write on a binary semaphore, which has only two possible values, zero or one. If the value is one, it means that there are no blocked processes in the semaphore structure. The execution of one or more *V* operations has no effect upon this one value. These semaphores are used for mutual exclusion, i.e., they allow only one process in a critical section at once. Milenković (1992) writes on general semaphores and presents the same definitions of the previous authors.

Deitel et al. (2004), Deitel (1984) and Milenković (1992) also changed the naming convention, using *wait* for *P* and *signal* for *V*. The *wait* operation, if there are no processes in the structure, allows the calling process to enter into its critical region. This is done after decrementing the semaphore value to 0, which means that the critical region is protected. Otherwise, it places the process in a blocking queue. This queue is exactly the difference that Reek (2002) considers absent in Dijkstra´s definition but definitely is present in the 1971 work by Dijkstra (1971) and which imposes a fair mechanism that avoids starvation.

The *signal* operation indicates that a process that was in its critical region now is outside, leaving it available. A process in the structure, if any, may now enter its critical region. A FIFO queuing discipline is assumed for the blocked processes.

This definition gives higher precedence to released processes than to new arrivals. It is named blocking-queue by Stark (1982), because the blocked processes are kept in a FIFO-queue instead of a set.

The Semaphore Metaphor

All of the previous definitions for semaphores are used for learning purposes. Those that increase the semaphore value, even in the presence of blocked processes in the same

semaphore, are better suited for the adoption of the analogous result of a traffic semaphore, a train cross-over signal, or a binary lock. These metaphorical figures are better understood by students, once they already familiar with how similar mechanisms work in the real world. But, as discussed earlier, those are weak definitions for the necessary mechanisms that obtain mutual exclusion and can produce starvation.

On the other hand, those definitions that do not change the semaphore value, when there is at least one process waiting the semaphore structure, sometimes are cumbersome to understand by the students, depending on the policy to choose what process will proceed. If the policy to choose is not fair, there is the possibility for processes stay indefinitely blocked in a semaphore. Those definitions can be also considered weak, although in a first sight seem to be strong.

Definitions that use a queue as the structure where processes wait in a semaphore are really the ones that are strong. And the queue metaphor could be used for learning purposes. Students know how works a civilized line of persons in the real world. This similar mechanism can be used to explain how processes stay in a semaphore with its zero value, and which process proceeds when a V operation is executed.

3. Semaphore Implementations

Reek (2002) alerts that if the semaphore mechanism provided by a given OS presents a different semantics than the one described in an adopted textbook, students trying to write and debug synchronization programs are likely to become confused.

Table 1 presents a comparison of four semaphore implementations, which are usually considered in laboratory assignments of introductory courses: POSIX (2004), System V (Dustan; Fris, 1995), Mac OS (Apple, 2009) and Win 3.2 (Microsoft, 2008).

Table 1. Characteristics of semaphore implementations.

	POSIX	System V	Mac OS	Win 3.2.
P naming	*sem_wait*	*semop*	*down*	*WaitOne*
Uses *busy-waiting* instructions	Yes	No	No	No
V naming	*sem_post*	*semop*	*up*	*Release*
V semantics	a one is added to the semaphore value and one of the processes, if existent, is removed from the structure	several processes may be released simultaneously	several threads may be released	several threads may be released
Released process must check the semaphore value again to continue its execution	Yes	depends on the scheduler strategy for the next process to be executed	Yes	Yes
Type	*weak*	partially weak	partially weak	partially weak
OS	Linux	Mac OS and Linux	Mac OS	Windows
Obs.	(1)	(2)	(3)	(3)

(1) According to Lewis and Berg (1998), a POSIX semaphore implementation uses another synchronization mechanism, i.e., *mutex*, in order to obtain mutual exclusion for the access to the

semaphore value.

(2) The *semop* system call can be used to specify many operations on a semaphore set through an array structure. Each operation is described by three inputs: an index, a value and some flags. The index defines one semaphore in the set and the value defines the semantics to be applied (*P* equivalent with a negative value or a *V* equivalent with a positive value). The OS tests whether or not all of the operations would succeed. An operation succeeds if the correspondent value added to the semaphore value would be greater than zero or if both the operation value and the semaphore value are zero. A zero or negative value means to wait (blocked) for the semaphore value to reach a positive value, but only if the set of flags do not point out that the system call is non-blocking. Every time a positive value is added, the OS must check if any of the blocked processes may now have applied its semaphore operations. If it is possible, the pending operations are applied and the process is released. This checking-to-release task is repeated, until no more operations can be applied. There is no information if this checking is based on the arrival order.

(3) Instead of processes, MacOS and Win 3.2 semaphores apply to threads.

A semaphore is considered partially weak because its implementation may cause starvation due to the amount of processes or threads that may be released when an equivalent V operation is executed, which requires that those released processes or threads compete among themselves and with new arrivals to enter a critical section.

Usually available semaphores in Linux are also available in other Unix OSs.

There is almost no compatibility between the four considered implementations and the reviewed definitions in OS introductory books. The definition by Silberschatz et al. (2004) and Stallings (2004) is the one that gets closer to the POSIX implementation.

If strong semaphores are desired, they will need to be implemented in top of an available implementation.

4. Conclusions

This paper discusses the problematic situation that happens when teaching students on synchronization and mutual exclusion. Three types of semaphores are presented together with a comparison of four of their implementations. OS teachers should be aware of what type of semaphores is defined in their adopted books and what are used in lab assignments in order to be able to clarify probable unexpected results.

Reek calls *strong semaphores* (Reek, 2002) those implementations that, upon the execution of a *V* operation or equivalent, do not need to recheck the semaphore value after their removal from the structure because the value was not modified. Thus a removed process has priority over new processes that attempt to decrease the value. Two of the three presented definitions are classified as strong, their difference is concerned to the strategy considered to release the blocked process. One grants fairness, the other not.

None of the reviewed semaphore implementations grant fairness, which is very difficult to achieve. Nevertheless, it seems interesting to search answers to the following questions: (a) Why are there differences in semaphore implementations? (b) Could fairness in solutions to theoretical and practical problems be guaranteed?

Those answers will be the focus of future work together with the analysis of other implementations.

References

Apple (2009) "Perl Programmers Reference Guide". http://developer.apple.com/documentation/Darwin/Reference/ManPages/man3/Thread__Semaphore.3pm.html, January.

Apt, K. R. (2002) "Edsger Wybe Dijkstra (1930–2002): A Portrait of a Genius (Obituary)", Formal Aspects of Computing, n. 14, p. 92–98.

Deitel, H. M. (1984) An Introduction to Operating Systems, Addison-Wesley.

Deitel, H. M.; Deitel, P. J.; Choffnes, D. R. (2004) Operating Systems (3rd Ed.), Prentice Hall.

Dijkstra, E. W. (1968) "The structure of the "THE"-multiprogramming system", CACM, v. 11, n. 5, p. 341 – 346.

Dijkstra, E. W. (1971) "Hierarchical Ordering of Sequential Processes". Acta Informática, v. 1, n. 2, p. 115-138.

Dustan, N.; Fris, I. (1995) "Process Scheduling and UNIX Semaphores". Software-Practice and Experience, v. 25, n. 10, p. 1141-1153. DOI: 10.1002/spe.4380251005

Lewis, B.; Berg, D. (1998) Multithreaded Programming with PThreads. Sun Microsystems Press.

Microsoft (2008) ".NET Framework Developer's Guide". http://msdn.microsoft.com/en-us/library/z6zx288a.aspx, September.

Milenković, M. (1992) Operating Systems, Concepts and Design (2nd Ed.), McGraw-Hill International Editions.

Posix (2004) "Portable Operating Systems and Interface for Unix", IEEE Standard 1003. 2004 ed. http://www.opengroup.org/onlinepubs/000095399/toc.htm, February.

Reek, K. A. (2002) "The Well-Tempered Semaphore: Theme with Variations". ACM SIGSE Bulletin, March, p. 356-359.

Silberschatz, A.; Galvin, P. B.; Gagne, G. (2004) Operating System Concepts (7th Ed.), Wiley.

Stallings, W. (2004) Operating Systems: Internals and Design Principles (5th Ed.), Prentice Hall.

Stark, E. W. (1982) "Semaphore primitives and starvation-free mutual exclusion". Journal of the ACM (JACM), v. 29, n. 4, p. 1049-1072.

Tanenbaum, A. S. (2008) Modern Operating Systems (3rd Ed.), Prentice Hall.

Tanenbaum, A. S.; Woodhull A. S. (2006) Operating Systems Design and Implementation (3rd Ed.), Prentice Hall.

THOCP. (2008) "The History of Computing Project". http://www.thocp.net, November.