Semaphore (programming)

In computer science, a **semaphore** is a variable or abstract data type that provides a simple but useful abstraction for controlling access by multiple processes to a common resource in a parallel programming environment.

A useful way to think of a semaphore is as a record of how many units of a particular resource are available, coupled with operations to *safely* (i.e., without race conditions) adjust that record as units are required or become free, and if necessary wait until a unit of the resource becomes available. Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called **counting semaphores**, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** (same functionality that mutexes have).

The semaphore concept was invented by Dutch computer scientist Edsger Dijkstra,[1] and the concept has found widespread use in a variety of operating systems.

Library analogy

Suppose a library has 10 identical study rooms, intended to be used by one student at a time. To prevent disputes, students must request a room from the front counter if they wish to make use of a study room. When a student has finished using a room, the student must return to the counter and indicate that one room has become free. If no rooms are free, students wait at the counter until someone relinquishes a room.

The librarian at the front desk does not keep track of which room is occupied, only the number of free rooms available. When a student requests a room, the librarian decreases this number. When a student releases a room, the librarian increases this number. Once access to a room is granted, the room can be used for as long as desired, and so it is not possible to book rooms ahead of time.

In this scenario the front desk represents a semaphore, the rooms are the resources, and the students represent processes. The value of the semaphore in this scenario is initially 10. When a student requests a room he or she is granted access and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the resulting value of the semaphore is negative,[2] they are forced to wait. When multiple people are waiting, they will either wait in a queue, or use Round-robin scheduling and race back to the counter when someone releases a room (depending on the nature of the semaphore).

Important observations

When used for a pool of resources, a semaphore does not keep track of which of the resources are free, only how many there are. Some other mechanism (possibly involving more semaphores) may be required to select a particular free resource.

Processes are trusted to follow the protocol. Fairness and safety are likely to be compromised (which practically means a program may behave slowly, act erratically, hang or crash) if even a single process acts incorrectly. This includes:

- requesting a resource and forgetting to release it
- releasing a resource that was never requested
- holding a resource for a long time without needing it
- using a resource without requesting it first.

Even if all processes follow these rules, *multi-resource deadlock* may still occur when there are different resources managed by different semaphores and when processes need to use more than one resource at a time, as illustrated by the dining philosophers problem.

Semantics and implementation

One important property of these semaphore variables is that their value cannot be changed except by using the wait() and signal() functions.

Counting semaphores are equipped with two operations, historically denoted as **V** (also known as signal()) and **P** (or wait())(see below). Operation **V** increments the semaphore **S**, and operation **P** decrements it. The semantics of these operations are shown below. Square brackets are used to indicate atomic operations, i.e., operations which appear indivisible from the perspective of other processes.

The value of the semaphore **S** is the number of units of the resource that are currently available. The **P** operation wastes time or sleeps until a resource protected by the semaphore becomes available, at which time the resource is immediately claimed. The **V** operation is the inverse: it makes a resource available again after the process has finished using it.

A simple way to understand wait() and signal() operations is:

- `wait()`: Decrements the value of semaphore variable by 1. If the value becomes negative, the process executing wait() is blocked, i.e., added to the semaphore's queue.
- `signal()`: Increments the value of semaphore variable by 1. After the increment, if the value was negative (meaning there are processes waiting for a resource), it transfers a blocked process from the semaphore's waiting queue to the ready queue.

Many operating systems provide efficient semaphore primitives that unblock a waiting process when the semaphore is incremented. This means that processes do not waste time checking the semaphore value unnecessarily.

The counting semaphore concept can be extended with the ability to claim or return more than one "unit" from the semaphore, a technique implemented in UNIX. The modified **V** and **P** operations are as follows:

```
function V(semaphore S, integer I):
    [S ← S + I]

function P(semaphore S, integer I):
    repeat:
        [if S >= I:
            S ← S - I
            break]
```

To avoid starvation, a semaphore has an associated queue of processes (usually a first-in, first out). If a process performs a **P** operation on a semaphore that has the value zero, the process is added to the semaphore's queue. When another process increments the semaphore by performing a **V** operation, and there are processes on the queue, one of them is removed from the queue and resumes execution. When processes have different priorities the queue may be ordered by priority, so that the highest priority process is taken from the queue first.

If the implementation does not ensure atomicity of the increment, decrement and comparison operations, then there is a risk of increments or decrements being forgotten, or of the semaphore value becoming negative. Atomicity may be achieved by using a machine instruction that is able to read, modify and write the semaphore in a single operation. In the absence of such a hardware instruction, an atomic operation may be synthesized through the use of a software mutual exclusion algorithm. On uniprocessor systems, atomic operations can be ensured by temporarily suspending preemption or disabling hardware interrupts. This approach does not work on multiprocessor systems where it is possible for two programs sharing a semaphore to run on different processors at the same time. To solve this problem in a multiprocessor system a locking variable can be used to control access to the semaphore. The locking variable is manipulated using a test-and-set-lock (TSL) command.

Example: Producer/consumer problem

In the Producer-consumer problem, one process (the producer) generates data items and another process (the consumer) receives and uses them. They communicate using a queue of maximum size N. Obviously the consumer has to wait for the producer to produce something if the queue is empty. Perhaps more subtly, the producer has to wait for the consumer to consume something if the buffer is full.

The problem is easily solved if we model the queue as a series of boxes which are either *empty* or *full*, and regard empty boxes as one type of resource and full boxes as another type of resource. The producer "removes" an empty box and then "creates" a full one, whilst the consumer does the reverse.

Given that `emptyCount` and `fullCount` are counting semaphores, and `emptyCount` is initially N whilst `fullCount` is initially 0, the producer does the following repeatedly:

```
produce:
    P(emptyCount)
    putItemIntoQueue(item)
    V(fullCount)
```

The consumer does the following repeatedly:

```
consume:
    P(fullCount)
    item ← getItemFromQueue()
    V(emptyCount)
```

Typically, either fullCount or emptyCount is set to 1, with the other set to 0. The producer or consumer that goes first will decrement the P value to 0, and after its critical section, it will increment the V value to 1. On the second iteration, it will block on the P, since it set it to 0 after grabbing it the first time, and the other process will run, releasing the first in the same way once it hits the V command at the end.

It is important to note that the order of operations is essential. For example, if the producer places the item in the queue *after* incrementing fullCount, the consumer may obtain the item before it has been written. If the producer places the item in the queue *before* decrementing emptyCount, the producer might exceed the size limit of the queue.

Function name etymology

The canonical names **P** and **V** come from the initials of Dutch words. **V** stands for *verhogen* ("increase"). Several explanations have been offered for **P**, including *proberen* for "to test,"[3] *passeer* for "pass," *probeer* for "try," and *pakken* for "grab." However, Dijkstra wrote that he intended **P** to stand for the portmanteau *prolaag*,[4] short for *probeer te verlagen*, literally "try to reduce," or to parallel the terms used in the other case, "try to decrease."[5][6][7] This confusion stems from the fact that the words for *increase* and *decrease* both begin with the letter *V* in Dutch, and the words spelled out in full would be impossibly confusing for those not familiar with the Dutch language.

In ALGOL 68, the Linux kernel,[8] and in some English textbooks, the **P** and **V** operations are called, respectively, **down** and **up**. In software engineering practice, they are often called **wait** and **signal**, **acquire** and **release** (which the standard Java library uses[9]), or **pend** and **post**. Some texts call them **procure** and **vacate** to match the original Dutch initials.

Semaphore vs. mutex

A mutex is essentially the same thing as a binary semaphore, and sometimes uses the same basic implementation. However, the term "mutex" is used to describe a construct which prevents two processes from accessing a shared resource concurrently. The term "binary semaphore" is used to describe a construct which limits access to a single resource.

In many cases a mutex has a concept of an "owner": the process which locked the mutex is the only process allowed to unlock it. In contrast, semaphores generally do not have this restriction, something the producer-consumer example above depends upon.

Notes and references

[1] Dijkstra, Edsger W. *Cooperating sequential processes (EWD-123)*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original (http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF); transcription (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html)) (September 1965)

[2] *The Little Book of Semaphores* (http://greenteapress.com/semaphores/) Allen B. Downey

[3] Silberschatz, Galvin & Gagne 2008, p. 234

[4] Dijkstra, Edsger W. *EWD-74*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original (http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF); transcription (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD00xx/EWD74.html))

[5] Dijkstra, Edsger W. *MULTIPROGAMMERING EN DE X8 (EWD-51)*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original (http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD51.PDF); transcription (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD00xx/EWD51.html)) (in Dutch)

[6] Dijkstra's own translation reads "try-*and*-decrease", although that phrase might be confusing for those unaware of the colloquial "try-and..." (http://www.wsu.edu/~brians/errors/try.html)

[7] (PATCH 1/19) MUTEX: Introduce simple mutex implementation (http://lkml.org/lkml/2005/12/19/34) Linux Kernel Mailing List, 19 December 2005

[8] Linus Kernel hacking HOWTO (http://www.linuxgrill.com/anonymous/fire/netfilter/kernel-hacking-HOWTO-5.html#ss5.3) LinuxGrill.com

[9] `java.util.concurrent.Semaphore`

- Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2008), *Operating System Concepts* (8th ed.), John Wiley & Sons. Inc, ISBN 978-0-470-12872-5

External links

- Dijkstra, Edsger W. *Over Seinpalen (EWD-74)*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original (http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF); transcription (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD00xx/EWD74.html)), Dijkstra introduces the concept (in Dutch)
- semaphore.h (http://www.opengroup.org/onlinepubs/009695399/basedefs/semaphore.h.html) programming interface - The Open Group Base Specifications Issue 6, IEEE Std 1003.1
- *The Little Book of Semaphores* (http://greenteapress.com/semaphores/) Allen B. Downey
- *A pragmatic, historically oriented survey on the universality of synchronization primitives* (http://www.oamk.fi/~joleppaj/personal/jleppaja_gradu_080511.pdf), Jouni Leppäjärvi

Article Sources and Contributors

Semaphore (programming) *Source*: http://en.wikipedia.org/w/index.php?oldid=491992409 *Contributors*: A.b.s, AJim, AKLowther, Aaron N. Tubbs, Adashiel, AeonicOmega, Agateller, Ahoerstemeier, Aidin 36, AlanH, Alca Isilon, Aldie, Alexius08, AllenDowney, Alue, Alvin-cs, Amit.bens, Andresoportus, Aravindh, Arsyed04, Artelius, Bemoeial, BenFrantzDale, Bender2k14, Bigdumbdinosaur, Borgx, Bradmarxmoosepi, C. A. Russell, CanisRufus, Carewolf, Carlox, Chevymontecarlo, Cowlinator, Crockett.jesse, CrookedAsterisk, Cybercobra, Daniel Dandrada, Daveh4h, Dead3y3, Diberri, Dilshan, Dori, Dysprosia, Eadric, Eaefremov, Ebraminio, Echoray, Edcolins, Edward, EdwardHades, Elias.k, Elipongo, EnOreg, Ferdinand Pienaar, Fg, Fram, Fwonce, Garde, Gazpacho, Gilliam, Glasreiniger, Haoao, HendrixEesti, Henrik, Hkmaly, Hobit, Hoxu, InlovewithGod, J.delanoy, JWHPryor, Jaimaganga, Jedonnelley, Jen savage, Jirka6, Joel7687, Joesmoe918, JonHarder, Jpp, Kbdank71, Kimjoarr, Kku, Kocsonya, Koreanjason, Krallja, Lakers, Lfstevens, Lingwitt, Loopy48, Maghnus, Magioladitis, Marinerg, Matt Cook, Medovina, Message From Xenu, Michael miceli, MikeAllen, Mild Bill Hiccup, Minesweeper, Mister-al-x, Mobius, MrTree, Mukulgupta281191, Nastytomato, Nikitadanilov, OMouse, Oceamus, Oli Filth, PerryTachett, Piano non troppo, Psyced, Pubuhan, Quuxplusone, R'n'B, R. S. Shaw, RTC Marine, RedWolf, Rholton, RickBeton, Robertwharvey, Rolypolyman, Roman12345, Rp, SJP, SalvoIsaja, Scgtrp, Shanes, Smelendez, Stupidenator, Tanner Swett, Tbhotch, Tburns, TedPavlic, Tedp, Texanos, Thecheesykid, Timwi, Tobias Bergemann, TomasS, Tseiff, Tsunanet, Uncle G, Unordained, Velco, Vfcrescini, Vinluvin, Visor, Wernher, Wikid77, Wikipelli, XP1, Xiaokehuang, Youneedaclue, 314 anonymous edits

License