**AOS Written Assignment 2: SEMAPHORES**

-Sai Sirisha Nadiminti (PG1, 2020201044)

**PART A: WORKING OF SEMPAHORES[[1]](#endnote-13002)**

A semaphore S is an integer variable that has two atomic operations, apart from initialization: wait() and signal(). The semaphores can be implemented with and without busy waiting.

When we implement the semaphores with busy waiting, the process waits till the value of semaphore becomes non-negative.

|  |  |
| --- | --- |
| Wait() | Signal() |
| wait(S) {  while (S <= 0); // busy wait  S--;  } | signal(S) {  S++;  } |

The wait() and signal() operations are atomic, I.e., when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S (S≤0), as well as its possible modification (S--), must be executed without interruption.

Rather than engaging in busy waiting, the process can block itself and go into waiting state. For this implementation, we need a waiting queue along with the integer value of semaphore. The waiting processes(their PCBs) are pushed to this waiting queue. The block() operation suspends the process that invokes it. A signal() operation removes one process from the list of waiting processes and awakens that process. The wakeup(P) operation changes the process from the waiting state to the ready state and resumes the execution of the process P. The process is then placed in the ready queue.

|  |  |  |
| --- | --- | --- |
| Definition of semaphore | Wait() | Signal() |
| typedef struct {  int value;  struct process \*list;  } semaphore; | wait(semaphore \*S) {  S->value--;  if (S->value < 0) {  add this process to S->list ;  block();  }  } | signal(semaphore \*S) {  S->value++;  if (S->value <= 0) {  remove a process P from S->list;  wakeup(P);  }  } |

The wait() and signal() operations are provided by the operating system as basic system calls.

If the value of the semaphore becomes –n (some negative value with value ‘n’), then it means that there are n processes in the waiting queue. The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use any queueing strategy.

In single-processor environment, we make the wait() and signal() operations atomic by inhibiting interrupts during the time the wait() and signal() operations are executing. In a multiprocessor environment, the SMP systems must provide locking techniques like compare and swap() or spinlocks for this purpose.

**PART B: PURPOSE OF THE SEMPAHORE**

The semaphore variable is used to solve critical section problems and to achieve process synchronization in the multi-processing environment.

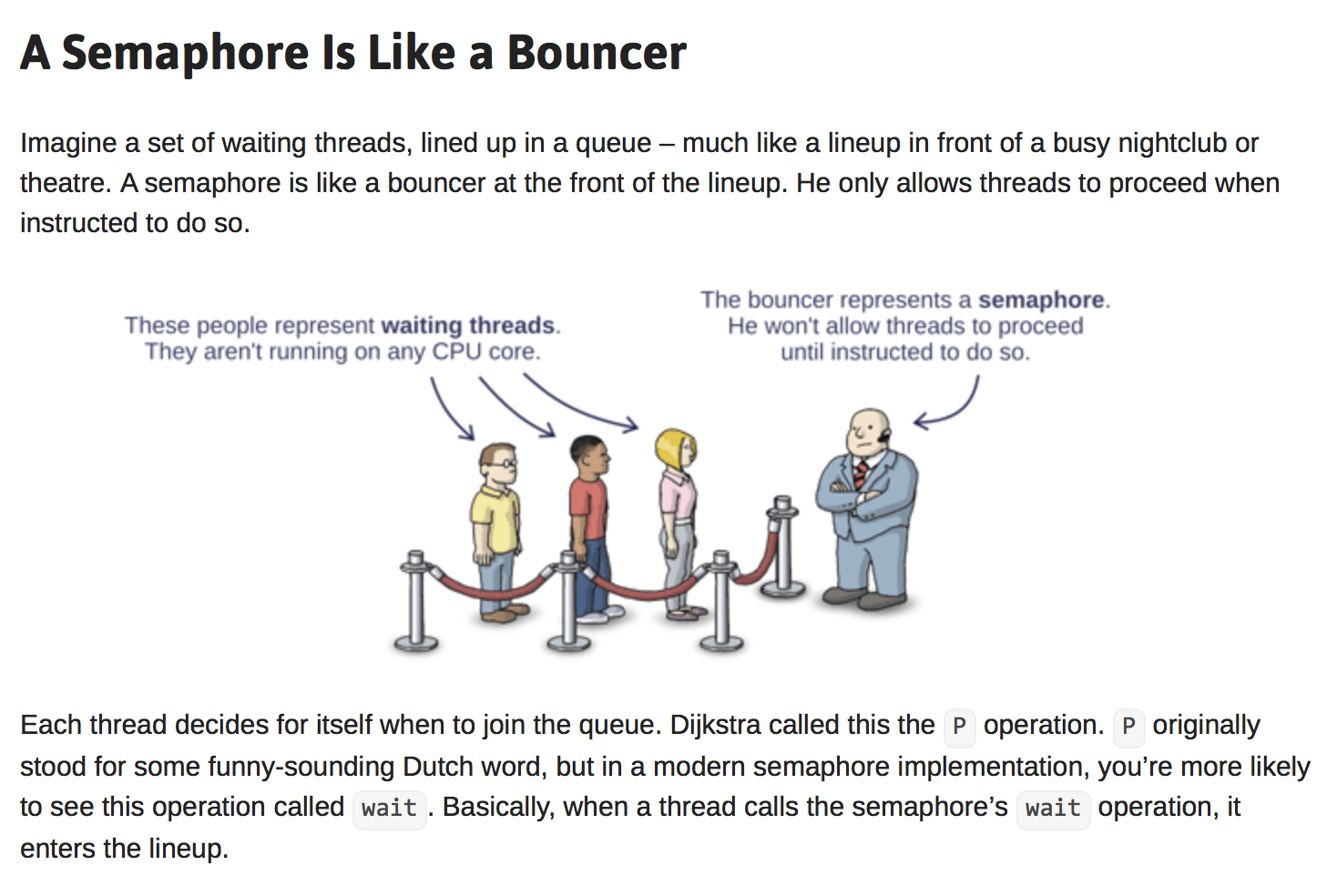


Fig 1. Purpose of the semaphore[[2]](#endnote-9853)

In operating systems, the semaphores are often distinguished into two types according to their purpose:

* 1. Counting semaphore - The value of a counting semaphore can take any integer value. This value represents the number of processes that can be executed simultaneously in the critical section. When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.
  2. Binary semaphore - The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. Binary semaphores can be used instead of providing mutual exclusion in the system that does not provide mutex locks.

**Using semaphores for process synchronization:**

We can use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 is completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0.

|  |  |
| --- | --- |
| P1 | P2 |
| S1;  signal(synch); | wait(synch);  S2; |

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

**PART C: Differences between semaphores, mutex and monitors:**

Process synchronization can be provided using mutex locks and monitors. Let us compare them to semaphores.

Difference between mutex vs semaphore: [[3]](#endnote-13666)

|  |  |
| --- | --- |
| Mutex | Semaphore |
| A mutex is a locking mechanism used to synchronize access to a resource. | Semaphore is signaling mechanism (“I am done, you can carry on” kind of signal). |
| Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. | Many tasks can acquire the semaphore in the case of counting semaphores. |
| The Mutex is an object. | Semaphore Is an integer value. |
| Mutex is only modified by the process that may request or release a resource. Mutex operations are lock or unlock. | The semaphore supports wait and signal operations. Wait() decrements the value of the semaphore and signal() increments the value of the semaphore. |
| A mutex provides mutual exclusion, which can be either a producer or consumer that can have the key (mutex) and proceed with their work. Till the producer fills the buffer, the consumer needs to wait, and vice versa. In Mutex lock, all the time, only a single thread can work with the entire buffer. | In the case of a single buffer, we can separate the 4 KB buffer into four 1 KB buffers. Semaphore can be associated with these four buffers. This allows users and producers to work on different buffers at the same time. |

Difference between semaphores and monitors:[[4]](#endnote-23259)

|  |  |
| --- | --- |
| Semaphore | Monitor |
| Semaphore is an integer variable S. | Monitor is an abstract data type. |
| The value of Semaphore S indicates the number of shared resources available in the system | The Monitor type contains shared variables and the set of procedures that operate on the shared variable. |
| When any process accesses the shared resources, it performs wait() operation on S and when it releases the shared resources it performs signal() operation on S. | When any process wants to access the shared variables in the monitor, it needs to access it through procedures. |
| The semaphore does not have condition variables. | Monitors have condition variables. |
| Semaphores are trickier to implement than monitors. | Monitors are easy to implement than semaphores. |
| In semaphores, mutual exclusion needs to be implemented explicitly. | Mutual exclusion in monitors is automatic. |
| Timing errors occur while using semaphores. | Monitors can overcome timing errors. |
| Shared variables are hidden in semaphores. | Shared variables are global to all processes in the monitor. |
| Semaphores permit more than one thread to access the critical section. | In monitors, mutual exclusion of threads is present. |

1. Operating System Concepts, Ninth Edition by [Avi Silberschatz,](http://www.cs.yale.edu/homes/avi) [Peter Baer Galvin,](http://www.galvin.info/) [Greg Gagne](http://people.westminstercollege.edu/faculty/ggagne) [↑](#endnote-ref-13002)
2. https://preshing.com/20150316/semaphores-are-surprisingly-versatile/ [↑](#endnote-ref-9853)
3. https://www.guru99.com/mutex-vs-semaphore.html [↑](#endnote-ref-13666)
4. https://www.geeksforgeeks.org/monitor-vs-semaphore/ [↑](#endnote-ref-23259)