1. Mutex can be released only by thread that had acquired it, while you can signal semaphore from any other thread (or process).

Mutex is used to protect the sensitive code and data, semaphore is more suitable for some synchronization problems like producer-consumer.

Unlike semaphores, mutexes provide priority inversion safety. Since the mutex knows its current owner, it is possible to promote the priority of the owner whenever a higher-priority task starts waiting on the mutex.

Mutexes also provide deletion safety, where the process holding the mutex cannot be accidentally deleted. Semaphores do not provide this.

2.sem

A pointer to the sem\_t object for the semaphore that you want to initialize.

pshared

Nonzero if you want the semaphore to be shared between processes via shared memory.

value

The initial value of the semaphore. A positive value (i.e. greater than zero) indicates an unlocked semaphore, and a value of 0 (zero) indicates a locked semaphore. This value must not exceed SEM\_VALUE\_MAX.

3. On processors supporting atomic compare-and-swap (Intel 486, Pentium and later, Alpha, PowerPC, MIPS II, Motorola 68k, Ultrasparc), the sem\_post and sem\_wait function is can safely be called from signal handlers.

4. The sem\_trywait() function attempts to decrement the value of the semaphore. The semaphore will be decremented if its value is greater than zero. If the value of the semaphore is zero, then sem\_trywait() will return -1 and set errno to EAGAIN.

The sem\_timedwait() function locks the semaphore referenced by sem as in the sem\_wait() function. However, if the semaphore can't be locked without waiting for another process or thread to unlock the semaphore by calling sem\_post(), the wait is terminated when the specified timeout expires. The timeout is based on the CLOCK\_REALTIME clock.