Threading – Part 2

Generally, thread execution must be ordered in order to prevent computation errors, such as ones that occur when reading and writing accesses to shared memory are unsynchronized.

Because threads may be scheduled at any time, a mechanism must be added to the system if we wish to impose ordering on thread execution.

Example of unsynchronized access: (taken from Bryant & O'Hallaron, 2nd Ed., p. 958; for a more recent, similar example, see 3rd Ed., p. 996.):

- thread-unsynchronized.c

A fundamental system resource for ordering execution are semaphores, which were created by Edsger Djikstra, who was Dutch.

A semaphore is a global variable that contains a nonnegative integer value.

It is used to guard a section of code in which synchronization is vital; thus, the section of code is known as a critical section.

The semaphore is manipulated by two methods, one for guarding access to the critical section and the other for releasing access.

The original names for these functions, in Dutch, were "Proberen", meaning "to test", and "Verhogen", meaning "to increment"; they are abbreviated simply as 'p' and 'v'.

The p and v functions were implemented in Unix as `sem_wait` and `sem_post`. A function was also added to initialize the semaphore, `sem_init`:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
 unsigned int value);
```

`pshared` determines whether the semaphore is shared between threads within a single process or between processes, using shared memory

```
int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */
```

`sem_wait` and `sem_post` are both atomic in operation, meaning that each function completes before any other calls to them can be started.

This atomic operation prevents more threads than allowed from entering a critical section simultaneously, as well as preventing unnecessary delays in access.

`sem_init` initializes the semaphore to a positive value.

`sem_wait` decrements the semaphore by 1 each time it is called.

When the value of the semaphore reaches 0, no other threads can enter the section; they must wait.

Hence, the value set by `sem_init` determines how many callers of `sem_wait` gain immediate, simultaneous entrance into the critical section.

Once a thread completes execution of the critical section, it must call `sem_post`, which increments the semaphore.

Once `sem_post` returns, an additional thread may enter the critical section.

Using a semaphore with just the values of 1 and 0 prevents simultaneous access to the critical section, also known as mutual exclusion.

A semaphore used for this purpose is known as a mutex.

A call to `sem_wait` is said to lock the mutex. While the thread is executing the critical section, it is said to be holding the mutex.

Once execution of the critical section is completed, the call to `sem_post` releases the mutex.

Example-- access made synchronized by mutex:

- thread-synchronized.c