KERNEL MASTERS

Linux System Programming

Synchronization

Authored and Compiled By: **Boddu Kishore Kumar**

Email: kishore@kernelmasters.org

Reach us online: www.kernelmasters.org

Contact: 9949062828

Important Notice

This courseware is both the product of the author and of freely available open source materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of this courseware cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - source code and binaries (where applicable) - that form part of this courseware, and that are present on the participant CD, are released under the GNU GPL v2 license and can therefore be used subject to terms of the afore-mentioned license. If you do use any of them, in any manner, you will also be required to clearly attribute their original source (author of this courseware and/or other copyright/trademark holders).

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant CD are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

1. Process Synchronization

Concurrent access to shared data may result in data inconsistency. We discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

We can show that the value of counter may be incorrect as follows. Note that the statement "counter++" may be implemented in machine language (on a typical machine) as

register1 = counter
register1 = register1 + 1
counter = register1

Where register1 is a local CPU *register*. Similarly, the statement "counter--" is implemented as follows:

register2 = counter
register2 = register2 - 1
counter = register2

Where again register2 is a local CPU register. Even though register1 and register2 may be the same physical register (an accumulator, say). The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution where the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is

T_0: *producer* execute $register_1 = counter$ $\{register_1 = 5\}$
T_1: *producer* execute $register_1 = register_1 + 1$ $\{register_1 = 6\}$
T_2: *consumer* execute $register_2 = counter$ $\{register_2 = 5\}$
T_3: *consumer* execute $register_2 = register_2 - 1$ $\{register_2 = 4\}$
T_4: *producer* execute $counter = register_1$ $\{counter = 6\}$
T_5: *consumer* execute $counter = register_2$ $\{counter = 4\}$

Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter == 6".

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the

particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way. Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Clearly, we want the resulting changes not to interfere with one another. Because of the importance of this issue, a major portion of this is concerned with **process synchronization and coordination**.

Problem in producer-consumer:

1. Inconsistency.
2. Loss of data.
3. Deadlocks

Race condition: Where several process access and manipulate the same data concurrently and the outcome of the execution depends on the particular order. It is called Race condition.

Critical section: Critical section is that part of the program where shared resources are accessed.

Critical-Section Problem:

Consider a system consisting of n processes {P0, P1, ..., Pn}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process P, is shown in Figure. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

```
do{
        entry section
        critical section
        exit section
        remainder section
} while (TRUE);
```

Pre-emption: A process gets executed in the middle of the other process.

No two processes are executing in their critical sections at the same time.

1.1. Synchronization Mechanisms:

To ensure mutual exclusion so that there is no problem of race condition among process.
Requirements of synchronization mechanisms:

1. **"Mutual Exclusion"** must always be guaranteed.
2. **"Progress":** If no process is executing in critical section and some process wish to enter their critical sections.
3. **"Bounded Waiting":** Bound or limit, no. of times that other processes are allowed to enter critical sections.

Two ways used to handle critical sections in OS.

Pre-emptive kernel	**Non pre-emptive kernel**
A process to be pre-empted while it is running in 'K' mode.	Doesn't allow a process running in 'K' mode to be pre-empted.
Most suitable for real time programming	Free from Race around conditions.

1.3. Semaphores:

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal (). The waitO operation was originally termed P (from the Dutch probercn, "to test"); signal () was originally called V (from verhogen, "to increment"). The definition of wait 0 is as follows:

```
wait(S) {
        while S <= 0
        ; // no-op
        S--;
}
```

The definition of signal () is as follows:

```
signal(S) {
        S++;
}
```

All the modifications to the integer value of the semaphore in the wait () and signal () operations must be executed indivisibly. That is, 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), and its possible modification (S—), must also be executed without interruption.

Operating systems often distinguish between **counting** and **binary** semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, mutex, initialized to 1.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a waitQ operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal () operation

(Incrementing the count). 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.

```
do {
        waiting(mutex);
        // critical section
        signal(mutex);
        // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with semaphores.

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096 byte length. A producer thread will collect the data and writes it to the buffer. A consumer thread will process the collected data from the buffer. Objective is, both the threads should not run at the same time.

Using Mutex:

A mutex provides mutual exclusion; either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa. At any point of time, only one thread can work with the entire buffer. The concept can be generalized using semaphore.

Using Semaphore:

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Misconception: (Difference between Binary semaphore and Mutex)

There is an ambiguity between binary semaphore and mutex. We might have come across that a mutex is binary semaphore. But they are not! The purpose of mutex and semaphore are different. May be, due to similarity in their implementation a mutex would be referred as binary semaphore.

Strictly speaking, a mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there will be ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is signaling mechanism (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend called you, an interrupt will be triggered upon which an interrupt service routine (ISR) will signal the call processing task to wakeup.

Toilet Example:

Mutex:

Is a key to a toilet. One person can have the key - occupy the toilet - at the time. When finished, the person gives (frees) the key to the next person in the queue.
Officially: "Mutexes are typically used to serialise access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section."(A mutex is really a semaphore with value 1.)

Semaphore:

Is the number of free identical toilet keys. Example, say we have four toilets with identical locks and keys. The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0. Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue.
Officially: "A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore)."

3.6. Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal () operation. When such a state is reached, these processes are said to be deadlocked.
To illustrate this, we consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

```
P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S);
  .                   .
  .                   .
  .                   .
signal(S);          signal(Q);
signal(Q);          signal(S);
```

Suppose that P0 executes wait (S) and then P1 executes wait (Q). When P0 executes wait (Q), it must wait until P0 executes signal (Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal (S). Since these signal () operations cannot be executed, P0 and P1 are deadlocked. We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. Another problem related to deadlocks is **indefinite blocking**, or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.