Module3_Part4

OPERATING SYSTEMS

Textbook : Operating Systems Concepts by Silberschatz

Semaphore Implementation

- The main **disadvantage** of the semaphore definition given here is that it requires busy waiting

  While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.

  This continual looping is clearly a problem in a real multiprogramming system where a single CPU is shared among many processes.

  Busy waiting wastes CPU cycles that some other process might be able to use productively.

Semaphore Implementation

- To overcome the need for busy waiting, we can modify the definition of

  the wait() and signal() semaphore operations.
- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.
- However, rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.

Semaphore Implementation

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

- To implement semaphores under this definition, we define a semaphore as a "C' struct:

```
typedef struct {
        int value;
        struct process *list;
        } semaphore;
```

Semaphore Implementation

Each semaphore has an integer value and a list of processes, list.

When a process must wait on a semaphore, it is added to the list of processes.

A signal() operation removes one process from the list of waiting processes and awakens that process.

- The wait() semaphore operation can now be defined as
- wait(semaphore *S) {

```
        S->value--;
        if (S->value < 0) {
        add this process to S->list;
        block();}
        }
```

Semaphore Implementation

- The signal () semaphore operation can now be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from  S->list;
                wakeup(P);
                }
        }
```

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Semaphore Implementation

- In this implementation, semaphore values may be negative, although semaphore values are never negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue.

Semaphore Implementation

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where

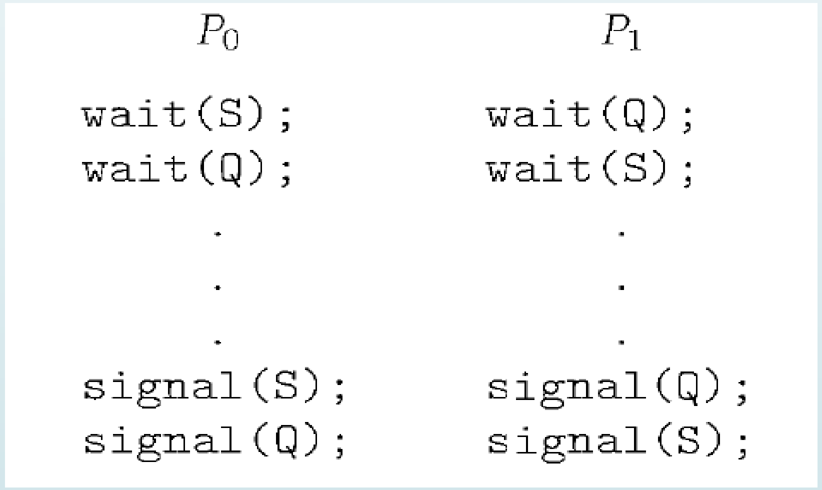
  two or more processes are waiting indefinitely for an event that can be caused only by one

  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

Deadlock and starvation

- To illustrate this, we consider a system consisting of two processes, *Po* 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 *Po* executes wait (S) and then P1 executes wait (Q). When *Po* executes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until *Po* executes signal(S). Since these signal() operations cannot be executed, *Po* and P1 are deadlocked.

Deadlock and starvation

- 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 is starvation--a situation in which processes wait indefinitely within the

  semaphore. Indefinite blocking may occur if we remove processes from the list associated

  with a semaphore in LIFO (last-in, first-out) order.

Priority inversion problem

Suppose there are three processes L(low priority process),M(Medium priority process) and H(High priority process)

- L is running in CS ; H also needs to run in CS ; H waits for L to come out of CS ; M interrupts L and starts running ; M runs till completion and relinquishes control ; L resumes and starts running till the end of CS ; H enters CS and starts running.
  Note that neither L nor H share CS with M.
- Here, we can see that running of M has delayed the running of both L and H. Precisely speaking, H is of higher priority and doesn’t share CS with M; but H had to wait for M. This is where Priority based scheduling didn’t work as expected because priorities of M and H got inverted in spite of not sharing any CS. This problem is called Priority Inversion.

Priority inheritance protocol

These systems solve the problem by implementing a *priority inheritance protocol*

According to this protocol, all processes that are accessing resources, needed by a higher-priority process, inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values.

In the example above, a priority-inheritance protocol would allow process *L* to temporarily inherit the priority of process H, thereby preventing process *M* from preempting its execution. When process *L* had finished using resource *R*, it would relinquish its inherited priority from H and assume its original priority. Because resource *R* would now be available, process H-not M-would run next.