Semaphores

Semaphores

- A semaphore S is an integer variable that, is accessed only through two standard atomic operations: wait() and signal().
- The wait() operation was originally termed P (from the Dutch *proberen,* “*to* test”).
- signal() was originally called V (from *verhogen,* “*to increment”).*

Semaphores

The definition of wait() is as follows:

```
wait(S)
 {
     while (S <= 0); // busy wait
      S--;
   }
```

The definition of signal() is as follows:

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

Semaphores

- when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In wait(S), the testing of the integer value of S ($S \leq 0$), and (S--), must be executed without interruption.

Semaphore Usage

Two types of Semaphores

Counting semaphore

- The value of a counting semaphore can range over an unrestricted domain.

Binary Semaphore

- The value of a binary semaphore can range only between 0 and 1.
- Thus, binary semaphores behave similarly to mutex locks.

Semaphore Usage

- 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 **wait()** 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.

Semaphore Usage

Semaphores can be used to solve various synchronization problems

- Consider two concurrently running processes: *P1 with a statement S1 and P2 with a statement S2.*
- *It requires that S2 be executed only* after *S1 has completed.*
- *Let us implement this scheme by letting P1* and *P2 share a common semaphore synch, initialized to 0.*

Semaphore Usage

- *In process P1, we* insert the statements

  S1;

  signal(synch);

- In process *P2, we insert the statements*

  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.*

Semaphore Implementation

- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows:
- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.
- Instead of 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

Semaphore Implementation

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

- 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.

Semaphore Implementation

The wait() semaphore operation can 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 be defined as

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

Semaphore Implementation

- 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

- Semaphore values may be negative , whereas 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.

- 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.

Semaphore Implementation

- We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time.

- We can solve it by simply disabling interrupts during the time the wait() and signal() operations are executing.

- This scheme works in a single-processor environment because, once interrupts are disabled, instructions from different processes cannot be interleaved.

- Only the currently running process executes until interrupts are reenabled and the scheduler can regain control

Semaphore Implementation

- Disabling interrupts on every processor can be a difficult task on a multiprocessor system.

- This scheme limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short (if properly coded, they should be no more than about ten instructions).

- Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time.

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.

- When such a state is reached, these processes are said to be **deadlocked.**

Deadlocks and Starvation

```
     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 *P1 executes signal(Q).*
- *Similarly,whenP1 executes wait(S), it must wait until P0 executes signal(S).*
- *Since these* signal() operations cannot be executed, *P0 and P1 are deadlocked*

Deadlocks and Starvation

- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.
- 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 remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Classical problems of synchronization

The Bounded-Buffer Problem

- The producer and consumer processes share the following data structures:

  int n;

  semaphore mutex = 1;

  semaphore empty = n;

  semaphore full = 0

- We assume that the pool consists of **n buffers**, each capable of holding one item.
- The **mutex semaphore** provides mutual exclusion for accesses to the buffer pool and is initialized to the value **1**.
- The empty and full semaphores count the number of empty and full buffers.
- The semaphore **empty** is initialized to the value **n**; the semaphore **full** is initialized to the value **0**.

The Bounded-Buffer Problem

```
do {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
   /* add next_produced to the buffer */
      . . .
   signal(mutex);
   signal(full);
} while (true);
```

The structure of the producer process.

The Bounded-Buffer Problem

```
do {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
} while (true);
```

The structure of the consumer process.