Operating system concepts

Process Synchronization (Semaphores, deadlocks)

Slides Set #11

By Prof K R Chowdhary

JNV University

2023

Semaphores are used to solve synchronization problems

- *Mutex* locks are the simplest of synchronization tools.
- *Semaphores*: are more robust, behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize
- A *semaphore S* is an integer variable, which is initialized, and accessed only through two standard *atomic* operations:
  1. **wait()**: "to test," (originally P: proberen)
  2. **signal()**: "to increment," (originally V: verhogen)
- Definition of wait():

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

- Definition of signal():

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

Semaphores usages: Example

- Let two concurrently running processes: P1 with a statement S1 and P2 with a statement S2 .
- We want that **S2 be executed only after S1 has completed**. In below code processes P1 and P2 share a common semaphore variable *sync*, which is initialized to 0.

```
//Process P1:
   S1;
   signal(sync);

//Process P2:
    wait(sync);
    S2;
```

- Questions:
  1. Which executes first, P1 or P2? Why?
  2. In above, if P1 is in loop of counter 1-5, how many times S1 is executed?
  3. In above, if P2 is in loop of 1-10, how many times S2 is executed?

Semaphores usages:

- There are two types of semaphores: *counting* and *binary semaphores*.
- The semaphore is initialized equal to the **number of resources available.** Each process that wishes to use a resource performs a wait() operation on the semaphore.
- When a process releases a resource, it performs a signal() operation.
- When the count for the semaphore goes to 0, all resources are in use.

Semaphores basic Questions:

1. What is a semaphore?
2. What is an atomic operations?
3. How the semaphore is busy-wait?
4. Suggest a real-life example of semaphore.
5. Which semaphore behaves like mutex lock (binary/counting)?
6. How many processes can be there in counting semaphore?
7. How many processes can be there in binary semaphore?
8. Which of the wait or signal semaphore is used for entry into process?
9. When the value of a semaphore is zero, what it indicates?
10. When a process acquires a resource, which semaphore (wait/signal) is executed?
11. When a process releases a resource, which semaphore (wait/signal) is executed?

Semaphore Implementation:

- The *mutex* locks suffers from *busy waiting*. The *wait()* and *signal()* semaphore present the same problem.
- To overcome the need for busy waiting, we can modify the definition of the *wait()* and *signal()* operations:
  - 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,
  - 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
  - Questions:
    1. How semaphore can be modified so that it does not consume cpu cycles due to busy-waiting?
    2. How a blocked process can be restarted from sleep?

Semaphore Implementation in Single processor system (without busy wait):

- Semaphore definition: Let *value* is initialized to 1.

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

- wait() semaphore operation definition, using above typedef:

```
wait(semaphore *S){
        S->value--;
        if (S->value < 0){  //no busy wait
           add this process at end of S->list;
           block();
        }
}
```

- The block() operation suspends the process that invokes it.
- This implementation may have semaphore values negative,

Semaphore Implementation in single processor system...

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

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

- 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.
- One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue,
- It is critical in that semaphore operations can be executed *atomically* (interrupts are disabled).

Semaphore Implementation in multiprocessor system:

- In a multiprocessor environment, *interrupts* must be disabled on every processor.
- SMP systems must provide alternative locking techniques – such as compare and swap() or spinlocks to ensure that wait() and signal() are performed atomically.
- It is important to admit that we have not completely eliminated busy waiting with this definition of the wait() and signal() operations.

Semaphores implemenation Questions:

1. Do the basic semaphore consume time due to busy waiting?
2. How to eliminate busy waiting in semaphore?
3. How to make a process sleep?
4. How to wake up a process that has gone into sleep?
5. What is meaning of semaphore value, say, −5 after executing "value−−"?
6. What is purpose of variable "list" in slide 8?
7. How a semaphore is implemented atomically?
8. How a semaphore is implemented in multiprocessor system?

Deadlock and starvation

- Situation where two or more processes are waiting indefinitely for an event that can be caused only by one of these waiting processes. When such a state is reached, these processes are said to be *deadlocked*.
- Processes P0, P1, each accessing two semaphores, S and Q implemented on single processor system, are set to value **1**:

```
   P0                      P1
wait(S);   <-------->  wait(Q);
wait(Q);               wait(S);
    ..                     ..
signal(S);             signal(Q);
signal(Q);             signal(S);
```

- Let P0 executes "wait(S)" and then P1 executes "wait(Q)".
- A set of processes is in a *deadlocked* state when every process in the set is waiting for an event to be caused only by another process in the set.
- Another problem related to deadlocks is *indefinite blocking* or *starvation*.

Priority Inversion:

- *Scheduling challenge*: Problem: A higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process
- Let there are three processes: L, M, H, whose priorities follow the order $L < M < H$. Assume that process H requires resource R, which is currently being accessed by process L.
- Suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority – process M – has affected how long process H must wait for L to relinquish resource R.
- In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H,
- This problem is known as *priority inversion*.

Dead lock Questions:

1. What is deadlock?
2. Give dead-lock example in real-life.
3. What is basic phenomena of occurrence of deadlock?
4. What is priority inversion? How it helps for removing deadlock?
5. What is starvation?
6. What is priority-inheritance protocol?

Classic Problems of Synchronization

There are number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme.

Bounded buffer problem:

- It is commonly used to explain the power of synchronization primitives. Following is the general structure of this scheme: Let the **producer** and **consumer** processes share the following data structures:

```
int n;  // n number of resources (buffers)
semaphore mutex = 1; // binary semaphore
semaphore empty = n; // no. of empty buffers
semaphore full = 0; // no. of full buffers
```

- Let there are *n* buffers, each can hold one item. The *mutex* semaphore (with initial value **1**) provides mutual exclusion for accesses to the buffers.

Classic Problems...: Bounded buffer problem

The structure of the **producer** process: P0

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

Classic Problems...: Bounded buffer problem...

The structure of the **consumer** process: P1

```
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);
```

Question: Produces/consumer increases/decrease full and increases/decreases empty.

Classic Problems: 1) Readers – Writers Problem

- Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to **read** the database, whereas others may want to update (that is, to **read** and **write**) the database.
- This problem is referred to as the readers – writers problem. Following is solution to the **first readers – writers problem**. Consider following data structures:

```
semaphore rw_mutex = 1; //binary
semaphore mutex = 1;//binary
int read_count = 0;//how many proc. are reading now
```

- The structure of a *writer* process: P0

```
do {
   wait(rw_mutex); // that is, update
   . . .
   /* writing is performed here */
   . . .
   signal(rw_mutex);
}
```

Classic Problems: Readers – Writers...

Code for a **reader** process: P1

```
do {
   wait(mutex);
   read_count++; // update read count
   if (read_count == 1) // one item is to be read
     wait(rw_mutex); //wait on rw_mutex if that is busy
   signal(mutex);  // it checks for count=1, and remains
                 //locked until reading is over (count=0)
      . . .
   /* reading is performed here */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Classic problem:

1. What is readers-writers problem?
2. Why deadlock occurs in reader-writers problem?
3. Can there be more that two processes that can use rw_mutex?
4. Why the variables in slide no. 17 are so initialized?

The Dining-Philosophers Problem

- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- The dining-philosophers problem is considered a classic synchronization problem

- One simple solution is to represent each chopstick with a semaphore.

```
semaphore chopstick[5];
do {
  wait(chopstick[i]);
  wait(chopstick[(i+1) % 5]);
  . . .
  /* eat for awhile */
  . . .
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);
  . . .
  /* think for awhile */
  . . .
} while (true);
```

The Dining-Philosophers Problem...

Several possible remedies to this deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available
- Use an asymmetric solution – that is, an odd-numbered philosopher picks up first his left chopstick and then his right chopstick, whereas an even-numbered does this in reverse,
- Questions:
  1. Why there is a deadlock in this problem?
  2. How the asymmetric solution works?
  3. There are still more solution(?)