Question 1

Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
    if (amount > highestBid)
        highestBid = amount;
}
```

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

Solution

A race condition is possible in this situation if two processes try to update the variable “highestBid” simultaneously. Suppose highestBid=1000 initially. Then two processes, process 1 and process 2, simultaneously execute bid(1300) and bid(1200) respectively. First both processes read the value of highestBid which is 1000. They both compare the value of highestBid with their respective amounts (1300 and 1200). The if condition is successful in both processes. Suppose process 1 updates highestBid first followed by process two. Then the final value of highestBid is 1200 whereas it should actually be 1300. This is because of the occurrence of race condition.

We can use mutex locks to prevent race conditions from occurring. The new code would look like the code below:

```
      lock.acquire();
      bid(amount);
      lock.release();
```

In the above code, we are protecting the variable “highestBid” using mutex lock.

Question 2

One approach for using compare and swap() for implementing a spin-lock is as follows:

```
void lock_spinlock(int *lock) {
    while (compare_and_swap(lock, 0, 1) != 0)
        ; /* spin */
}
```

A suggested alternative approach is to use the “compare and compare-and-swap” idiom, which checks the status of the lock before invoking the compare and swap() operation. (The rationale behind this approach is to invoke compare and swap()only if the lock is currently available.) This strategy is shown below:

```
void lock_spinlock(int *lock) {
    while (true) {
        if (*lock == 0) {
            /* lock appears to be available */

            if (!compare_and_swap(lock, 0, 1))
                break;
        }
    }
}
```

Does this “compare and compare-and-swap” idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

Solution

This alternate implementation of a spinlock is correct and effective.

The function repeatedly checks the value of the lock variable in a busy-wait loop. If the lock appears to be available (i.e., its value is 0), the function attempts to acquire the lock using an atomic compare-and-swap operation.

The compare-and-swap operation tries to update the value of the lock variable from 0 to 1 atomically. If the operation succeeds (i.e., the value of the lock was still 0 when the compare-and-swap was executed), the function breaks out of the loop and returns, indicating that the lock has been acquired. If the operation fails (i.e., another thread acquired the lock before this thread), the loop continues.

Question 3

Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Illustrate how semaphores can be used by a server to limit the number of concurrent connections.

Solution

Semaphores can be used by a server to limit the number of of concurrent connections as follows:

```
semaphore n_connections = N;
void make_connection(){
      wait(n_connections);
      /* make a connection */
}

void close_connection(int i){
      signal(n_connections);
      /* close connection i */
}
```

In the above code, we have a semaphore with value initialized to N, the maximum number of allowed connections. A connection can be established only if the value of the semaphore is at least 1. Everytime a connection is established, the semaphore is decremented by 1. Everytime a connection is closed, the semaphore is incremented. This ensures that at most N connections are made concurrently.

Question 4

In the class we discussed a solution for solving the readers-writers problem. But it can cause the writer threads to starve. Can you develop a solution in which no threads starve?

Solution

In the readers-writer problem, the writer gets starved when multiple reader threads keep entering the critical section which prevents the writer thread from entering the critical section. To prevent this, we need to implement a first come first served basis of allocating resources. We can assume that semaphore queues are fair. Thus a starvation free implementation can be done as follows:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
semaphore sem = 1;

void writer(){
      wait(sem);
      wait(rw_mutex);
      /* writing is performed */
     signal(sem);
      signal(rw_mutex);
}

void reader(){
      wait(sem);
     wait(mutex);
      read_count++;
      if(read_count==1) wait(rw_mutex);
      signal(mutex);
      signal(sem);
      /* reading is performed */
      wait(mutex);
      read_count--;
      if(read_count==0) signal(rw_mutex);
      signal(mutex);
}
```

In the above solution, suppose a writer thread requests semaphore “sem” and after that a reader thread arrives. The writer thread will first acquire semaphore “sem”. Thus the reader thread will not be able to acquire rw_mutex. Thus whenever currently executing readers stop reading, the writer will start writing.

In this solution, whichever thread comes first will first acquire semaphore “sem” and thus will be able to execute first. Thus it is a starvation free solution.

Question 6

In the class we discussed atomic increment operation implemented using CAS instructions. But that implementation does not ensure guaranteed termination of each method, i.e. starvation-freedom. Can you modify this method so that the modification ensures that every thread invoking increment method will eventually terminate.

Solution

Starvation can happen in the atomic increment implementation using CAS if a thread does not get a chance to increment the variable because other threads keep updating the variable. We can overcome this problem by using a semaphore with a fair queue as follows:

```
semaphore s;

void increment(atomic_int *v){
     int temp;
     do {
          temp = *v;
     }
     wait(s);
     while (temp != compare_and_swap(v, temp, temp+1));
     signal(s);
}
```

Since the semaphore is fair, each thread will get a fair chance to increment the variable and thus there will be no starvation.