Testing Concurrent Programs -17, Exercise after third lecture

At the "Lab" session on April 11, we can have a look at some exercises including the following ones.

1. Implementing a Counting Semaphore

This exercise is based on an some suggested (flawed) implementations of counting semaphores that appeared in textbooks on operating systems (in the previous century). In this homework, two implementations will be described. Your task is to model these implementations in Promela, analyze them in SPIN to find out the flaw, explain the flaw, and finally to suggest a fix.

First some recollection of semaphores. A *counting semaphore* is an object, whose internal state can be thought of as a nonnegative integer, with two operations.

- A *signal* operation (denoted V) atomically increments the value of the integer.
- A *wait* operation (denoted P) atomically decrements the value of the integer if it is positive, and blocks if it is 0. The operation can be resumed if another thread performs a V operation.

One use for counting semaphores is to synchronize producers and consumers that share some data structure (e.g., a buffer of produced items), where a producers peforms a signal operation after inserting an item, and a consumer performs a wait operation before retrieving an item.

A *binary semaphore* has an internal state which is either 0 or 1, with operations *binary signal* (VB) and *binary wait* (PB), such that

- VB sets the state to 1,
- PB atomically decrements the value of the semaphore if it is 1, otherwise it blocks (waiting for another VB operation).

The below algorithms all implement a counting semaphore, using binary semaphores as a given primitive. Each operation on the counting semaphore is a nonatomic sequence of statements, and it should appear as if the corresponding operation is performed atomically at some point in time between

the invocation and return of the operation. It is sometimes not trivial to identify this point, but one can at least derive some minimal requirements, such as the following, assuming that the initial value of the semaphore is 0.

- It should not be possible to perform more completed P operations than initiated V operations.
- If m V operations have completed, then it should be possible to perform at least m P operations (i.e., it should not be the case that some of them block).

Such requirements can be checked by suitable scenarios with suitable sets of threads performing P operations, and suitable sets of threads performing V operations.

Now over to the algorithms. Here is the first one. It assumes a declaration of the semaphore state as a structure

```
struct semaphore {
  binsem mutex = 1;
  binsem delay = 0;
  int count = 0;
}
```

where `binsem` denotes a binary semaphore. The two operations are given as follows in C-like pseudocode

```
void P(semaphore s) {
  PB(s.mutex);
  s.count = s.count − 1;
  if (s.count < 0) {
    VB(s.mutex);
    PB(s.delay);
  }
  else VB(s.mutex);
}
```

```
void V(semaphore s) {
  PB(s.mutex);
  s.count = s.count + 1;
  if (s.count <= 0) VB(s.delay);
  VB(s.mutex);
}
```

The second algorithm uses a slightly different data structure

```
struct semaphore {
  binsem mutex = 1;
  binsem delay = 0;
  int count = 0;
  int wakecount = 0;
}
```

and thereafter the two operations are given as

```
void P(semaphore s) {
  PB(s.mutex);
  s.count = s.count − 1;
  if (s.count < 0) {
    VB(s.mutex);
    PB(s.delay);
    PB(s.mutex);
    s.wakecount = s.wakecount − 1;
    if (s.wakecount > 0) VB(s.delay);
  }
  VB(s.mutex);
}
```

```
void V(semaphore s) {
  PB(s.mutex);
  s.count = s.count + 1;
  if (s.count <= 0) {
    s.wakecount = s.wakecount + 1;
    VB(s.delay);
  }
  VB(s.mutex);
}
```

2. Tricky Queue by Herlihy and Wing

A cute concurrent algorithm that can be modeled in SPIN is the queue by Herlihy and Wing (Herlihy and Wing 1990), which is very small and looks simple, but is actaully quite tricky to understand correctly.

The algorithm operates on an array `items[SIZE]` of data values (of type *value*, say), and has a pointer `back` to the first non-used element. `items` is initialized with `NULL` values. The two operations are given as follows in C-like pseudocode

```
void EnQueue(value val) {
  i = INC(back);
  items[i] = val;
}
```

```
void DeQueue() {
  while true do;
    range = back;
    for i = 0 to range do;
      x = SWAP(items[i],NULL);
      if x != NULL then return x;
}
```