SWEN90004
Modelling Complex Software Systems

Semaphores and State Diagrams

Nic Geard

Lecture Con.03

Semester 1, 2019

Making concurrency easier...

Going through the mutex algorithms is a useful lesson, as it shows the various issues and pitfalls involved in securing mechanisms for mutual exclusion. However, the correct algorithms are somewhat complex and tedious to implement.

Semaphores provide a concurrent programming construct on a higher level than machine instructions.

Using semaphores, the critical section problem can be solved trivially.

Semaphores

A semaphore is a simple but versatile concurrent device for managing access to a shared resource.

It consists of a value $v \in \mathbb{N}$ of currently available access permits, and a wait set W of processes currently waiting for access.

It must be initialized $S := (k, \{\ \})$, where k is the maximum number of threads can simultaneously access some resource.

S comes with two atomic operations, *wait* and *signal*.

Semaphore operations

Assume process *p* executes the operation:

`S.wait()`

```
if S.v > 0
    S.v--
else
    S.W = S.W U {p}
    p.state = blocked
```

`S.signal()`

```
if S.W == { }
    S.v++
else
    choose q from S.W
    S.W = S.W \ {q}
    q.state = runnable
```

Hence when *p* signals *S* whose wait set is empty, *S*'s value is incremented; otherwise an arbitrary process is unblocked, that is, removed from the wait set, having its state changed to runnable.

The binary semaphore

If $S.v \in \{0, 1\}$, S is called binary.

For a binary semaphore the operations are:

`S.wait()`

```
if S.v == 1
    S.v = 0
else
    S.W = S.W U {p}
    p.state = blocked
```

`S.signal()`

```
if S.W == { }
    S.v = 1
else
    choose q from S.W
    S.W = S.W \ {q}
    q.state = runnable
```

If $S.v = 1$ then `S.signal()` is undefined.

Sometimes a binary semaphore is called a mutex.

The mutex problem again

Using a binary semaphore, it is easy to solve the mutex problem for two processes:

```
binary semaphore S = (1,{});
```

```
loop
p1: non_critical_p();
p2: S.wait();
p3: critical_p();
p4: S.signal();
```

```
loop
q1: non_critical_q();
q2: S.wait();
q3: critical_q();
q4: S.signal();
```

State diagrams

State diagrams are often used to describe the behaviour of systems.

They are directed graphs, where nodes represent states and edges represent transitions, that is, state changes.

A state gives pertinent information about a process at a given point in time—usually the values of its instruction pointer and local variables.

The mutex problem again

We will consider an “abbreviated” form of the mutex algorithm, in which we absorb the `non_critical` and `critical` sections into the following `wait` and `signal` statements, respectively.

```
binary semaphore S = (1,{});
```

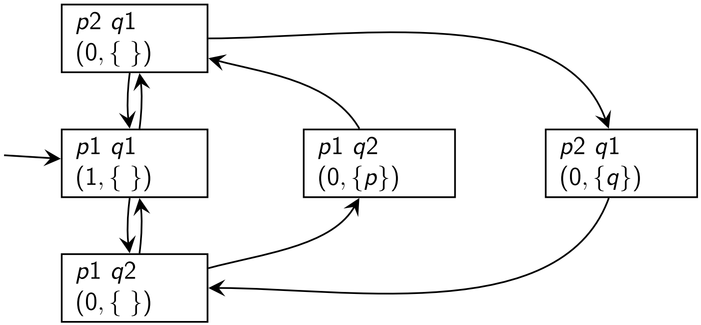
```
loop
p1: S.wait();
p2: S.signal();
```

```
loop
q1: S.wait();
q2: S.signal();
```

The state is given by the tuple (pi, qi, S) where pi and qi are P and Q's instruction pointers, respectively, and S is the state of the seamphor (v, W) where v is the number of threads available to access the resource and W is the set of waiting processes.

State diagram for the semaphore solution

Again, we just use the four interesting program points:



$$\begin{tikzcd}
 & \boxed{\begin{matrix} p2\ q1 \\ (0, \{\ \}) \end{matrix}} \arrow[d, bend right] \arrow[drr, bend left] & & \\
{} \arrow[r] & \boxed{\begin{matrix} p1\ q1 \\ (1, \{\ \}) \end{matrix}} \arrow[u, bend right] \arrow[d, bend right] & \boxed{\begin{matrix} p1\ q2 \\ (0, \{p\}) \end{matrix}} \arrow[ul, bend right] & \boxed{\begin{matrix} p2\ q1 \\ (0, \{q\}) \end{matrix}} \arrow[dll, bend left] \\
 & \boxed{\begin{matrix} p1\ q2 \\ (0, \{\ \}) \end{matrix}} \arrow[u, bend right] \arrow[ur, bend right] & &
\end{tikzcd}$$

The diagram shows that the solution is correct, and that there is no deadlock or starvation.

Using semaphores to control execution order

```
integer array A
binary semaphore S1 = (0,{})
binary semaphore S2 = (0,{})
```

```
p1: sort low half
p2: S1.signal()
p3:
```

```
q1: sort high half
q2: S2.signal()
q3:
```

```
m1: S1.wait()
m2: S2.wait()
m3: merge halves
```

Strong semaphores

The binary semaphore solution to the mutex problem generalises to N processes.

However, when $N > 2$, there is no longer a guarantee of freedom from starvation, because blocked processes are taken arbitrarily from a set.

The obvious (fair) way of implementing semaphores is to let processes wait in a queue.

This removes the starvation issue and we then talk about strong semaphores.

The bounded buffer problem

Assume we have a “producer” process p and a “consumer” process q.

Process p generates items for q to process. If the two are of similar average speed, but each of varying speed, then the use of a buffer can smoothen overall processing and speed it up, allowing for asynchronous communication between the two.

General semaphores can be used to implement the cooperation. The idea is to have two semaphores $S1$ and $S2$ and maintain a loop invariant $S1.v + S2.v = n$, where n is the buffer size.

Because of the roles they play, let us call the semaphores `notEmpty` and `notFull`.

Semaphores for the bounded buffer problem

```
buffer = empty queue;
semaphore notEmpty = (0,{});
semaphore notFull = (n,{});
```

```
item d
loop
p1: d = produce();
p2: notFull.wait();
p3: buffer.put(d);
p4: notEmpty.signal();
```

```
item d
loop
q1: notEmpty.wait();
q2: d = buffer.take();
q3: notFull.signal();
q4: consume(d);
```

Semaphores in Java

The package `java.util.concurrent` has a `Semaphore` class.

The wait and signal operations are called `acquire()` and `release()`.

The `Semaphore` constructor has, apart from the value argument, an optional Boolean argument which, when true, makes the semaphore strong; that is, it gives access to waiting threads on a "first in, first out" basis.

Summary

- Sempahores are an elegant and efficient construct for solving problems in concurrent programming.
- Semaphorse are widely implemented.
- Liveness properties of semaphores may depend on implementation.
- Monitors (next lecture) make concurrency easier still.

References

- M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice Hall, 2nd edition, 2006.
- C. A. R. Hoare, Monitors: An operating system structuring concept, *Communications of the ACM*, 17(10):549–557, 1974.
- D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 2nd edition, 2000.