CS766: Analysis of concurrent programs 2023

Lecture 7: Thinking concurrency

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2023-01-28

Topic 7.1

Concurrency

We all have multicore machines

1. All on same chip
2. Shared memory
3. Number of cores are ever increasing

Cores were bigger and programs ran faster

Moore's law: every year we bought faster computers.

Cores were bigger and programs ran faster

Moore's law: every year we bought faster computers.

More cores and programs got stuck...

- Speedup is not is sublinear
- Cores are waiting on each other
- Synchronization needs a careful design

Sequential vs concurrent

- One processor is working on memory
- Multiple processor is working on shared memory

Asynchronous

Unpredictable delays

- Chache misses
- Page fault
- Waiting to be scheduled
- killed process

Exercise 7.1

What are the expected delays in the above conditions?

Concurrency jargon

We will consider the following synonymous.

- Processors
- Threads
- Process

Topic 7.2

An example

Example: concurrent primality testing

- Objective : print primes from 1 to 10^{10}
- Resources : 10 processors; one thread per processor
- Goal: Get 10 fold speedup

EvenPrime: divide the load in equal parts

- Each thread tests range of 10^9

```
void evenPrime {
  int i = thread.getId(); // IDs in {0..9}
  for( int j = i*10^9 +1, j<(i+1)*10^9 ; j++ ) {
    if( isPrime(j) )
      print(j);
  }
}
```

- Higher ranges have fewer primes and larger numbers harder to test
- Thread workloads: actually uneven and hard to predict
- Need dynamic load balancing

DynamicPrime: free threads get a number

```
int counter = 1;         // global - lives in shared memory

// code of each thread
void dynamicPrime {
  long j = 0;
  while (j < 10^10 ) {  // stop when all values taken
    j = counter++;
    if( isPrime(j) )
       print(j);
  }
}
```

`counter++` is not atomic.
Does not work for concurrent threads

Exercise 7.2

Will the above really stop at 10^9*?*

Implementation of `counter++`

Running in each thread:

```
tmp = counter;    // global read
counter = tmp+1;  // global write
j = tmp;          // local
```

An undesirable execution:

```
thread0: Rcounter=1
thread1: Rcounter=1
thread1: Wcounter=2
thread0: Wcounter=2
```

Notation of Memory events:

```
R/W<VariableName>=<Value>
```

A hardware fix of the problem.

If somehow we can glue the following read and write events.

```
tmp = counter;    // global read
counter = tmp+1; // global write
```

Modern processor provide `ReadModifyWrite` instruction for this task.

Algorithmic mutual exclusion

Let us suppose we do not have such an instruction.

We need to ensure that the processes do not interfere each other.

Let us develop ideas that allow us to implement mutual exclusion without special instruction.

Topic 7.3

The fable - by Maurice Herlihy and Nir Shavit

The fable: Alice and bob share a pond

- Both Alice and Bob have pets
- Pets hate each other

Two kind of requirements

- Safety : nothing bad happens
  - Both pets never in the pond at the same time. Mutual exclusion
- Liveness: something good eventually happens
  - If only one wants in, it gets in starvation-free
  - If both wants in, one gets in deadlock-free

Simple protocol

Protocol:

- Look at the pond
- Release pet in the pond

Issues:

- Looking and release are not atomic
- They may not be fully visible to each other.

Lessons:

- Threads cannot see internal actions of each other.
- Explicit communication required

Phone protocol

Protocol:

- They call each other.
- Only after a permission, they put their pet in.

Issues:

- Bob is taking shower when Alice called
- Alice may be dead when Bob called.

Lessons:

- Recipient thread may be busy or killed
- Communication must be persistent (like writing) not transient (like talking)
- Message passing does not work

Can protocol

Protocol:

- Cans on the windowsill of Alice
- Strings attached to the cans go to Bob's house
- Bob pulls the string and knocks over the cans.

Issues:

- Cans cannot be used again(why?)
- Bob runs out of cans.

Lessons:

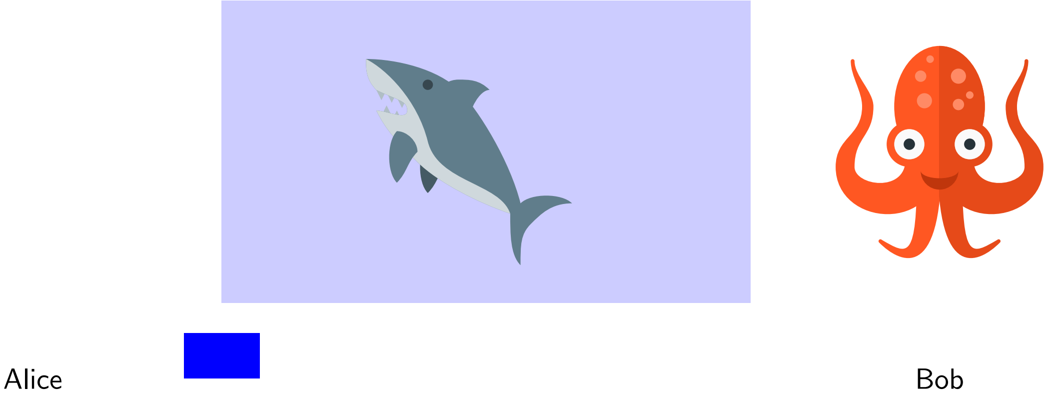
- Interrupts (sender sets and receiver resets when ready) cannot solve mutual exclusion
- Needs unbound interrupt bits

Flag protocol: both have flags

- They can raise and lower their flags.
- Both can see each other's flags.

Flag protocol: access for alice

1. Raise flag
2. Wait until Bob's flag is down
3. Release pet
4. Lower flag when pet returns



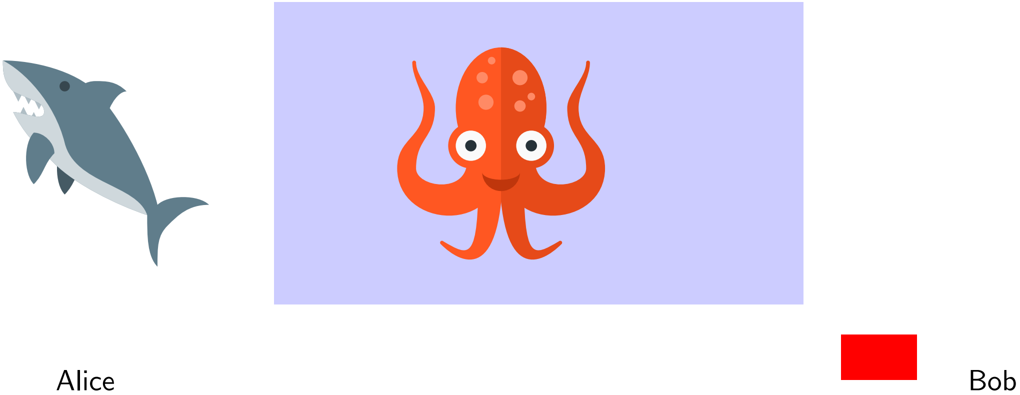
Flag protocol: access for Bob

1. Raise flag
2. Wait until Alice's flag is down
3. Release pet
4. Lower flag when pet returns

Deadlock possible

Exercise 7.3

Can we swap instructions 1 and 2 in both or in one?



2nd Flag protocol for Bob to avoid deadlock

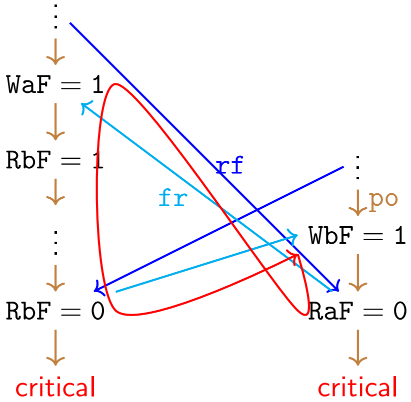
1. Raise flag
2. While Alice's flag is up
    - Lower flag
    - Wait for Alice's flag to go down
    - Raise flag
3. Unleash pet
4. Lower flag when pet returns

Alice has priority. Starvation for Bob is possible

Proving mutual exclusion in 2nd flag protocol

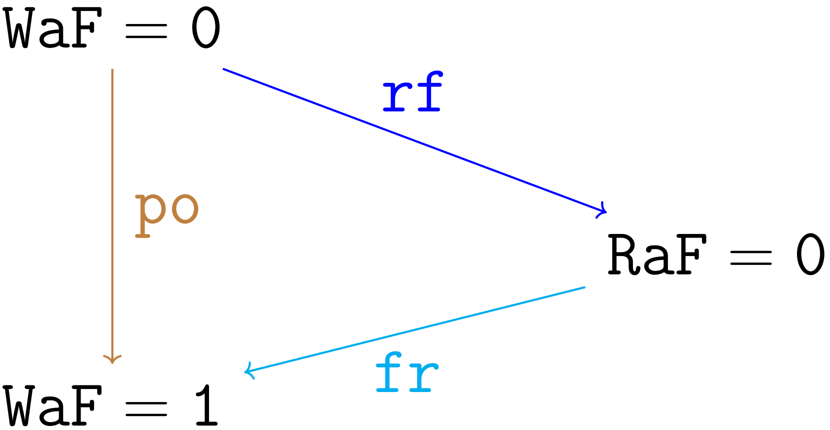
```
pre: aF := bF := 0

                              thread Bob:
                              b1:bF := 1
thread Alice:                 b2:while(aF==1){
a1:aF = 1                     b3:  bF := 0
a2:while(bF==1);          ||  b4:  while(aF==1);
a3:..// critical              b5:  bF := 1
a4:aF := 0                    b6:}
                              b7:..//critical
                              b8:bF := 0
```

Violating execution:
Assume threads reached critical section at the same time.



There cannot be a cycle in the trace drawing.
Therefore, the trace is impossible.

What is `fr`?

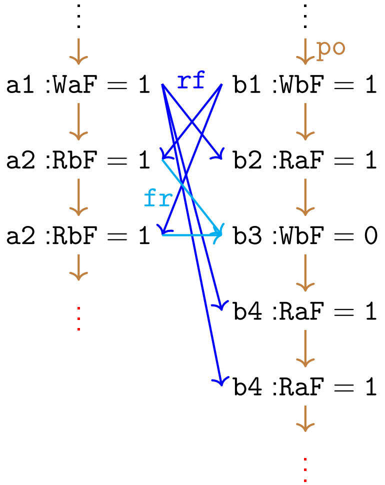


Proving deadlock freedom in 2nd flag protocol

```
pre: aF := bF := 0
```

```
                          thread Bob:
                          b1:bF := 1
thread Alice:             b2:while(aF==1){
a1:aF = 1                 b3:  bF := 0
a2:while(bF==1);     ||   b4:  while(aF==1);
a3:..// critical          b5:  bF := 1
a4:aF := 0                b6:}
                          b7:..//critical
                          b8:bF := 0
```

Violating execution: Assume both threads are stuck in lassos



b3 cannot happen after infinitely many events a2,a2,...

End of Lecture 7