ISCTE IUL
Instituto Universitário de Lisboa

Coordination Problems

Programação Concorrente e Distribuída

Parallel and Distributed Programming

2015-2016

Deadlocks, Livelocks and Starvation
Problems with coordination

Philosophers' dinner,
Readers-writer's problem
Cigarette smokers' problem
Santa Clause problem
Building H2O problem
River crossing problem

Deadlock

Deadlock is when a application is in a state where two or more threads are each waiting for each other to release a resource.

This is the main and most difficult problem to solve in concurrent programming.

Livelock

A **livelock** is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another.

Starvation

Starvation happens when a thread is not able to gain access to a shared resource.

This can happen when critical sections are big, forcing other threads to wait and reducing the level of concurrency.

Dining Philosophers Problem

Dijkstra proposed the Dining Philosophers Problem in 1965. It appears in a number of variations, but the standard features are a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti. Five philosophers, who represent interacting threads, come to the table and execute the following loop:

```
while(true){
    think();
    get_forks();
    eat();
    put_forks();
}
```

The forks represent resources that the threads have to hold exclusively in order to make progress. The thing that makes the problem interesting, unrealistic, and unsanitary, is that the philosophers need two forks to eat, so a hungry philosopher might have to wait for a neighbor to put down a fork.

Assuming that the philosophers know how to think and eat, our job is to write a version of get forks and put forks that satisfies the following constraints:

- Only one philosopher can hold a fork at a time.
- It must be impossible for a deadlock to occur.
- It must be impossible for a philosopher to starve waiting for a fork.
- It must be possible for more than one philosopher to eat at the same time.

The last requirement is one way of saying that the solution should be efficient; that is, it should allow the maximum amount of concurrency.

Dining Philosophers Problem

Proposed by Dijkstra in 1965

- 5 philosophers
- 5 forks

Philosophers:

```
while(true){
   think();
   get_forks();
   eat();
   put_forks();
}
```

Dining Philosophers Problem

- Only one philosopher can hold a fork at a time.
- It must be impossible for a deadlock to occur.
- It must be impossible for a philosopher to starve waiting for a fork.
- It must be possible for more than one philosopher to eat at the same time.

Solutions to the Philosophers Dinner

Use specific order to get the forks

- Try to implement the a solution that uses the following approach:
  1. try to take the first fork; if the fork is not available then wait;
  2. try to take the second fork; if the fork is not available then wait.
  3. when a philosopher puts down a fork, it wake up (notifies) any other philosopher that may be interested in picking up the fork.

Fork.java

```
class Fork{
	private int id;
	private boolean inUse;
	public Fork(int id){
		this.id=id;
		inUse=false;
	}
	public synchronized void up(){
		while(inUse){
		   System.out.println(“Going to sleep waiting ("+id+")");
		    try{
		      wait();
		    }catch(InterruptedException e){e.printStackTrace();}
		}
		System.out.println("Fork("+id+") up");
		inUse=true;
	}
	public synchronized void down(){
		System.out.println("Fork("+id+") down");
		inUse=false;
		notifyAll();
	}
}
```

Philosopher.java

```
class Philosopher extends Thread {
	int id,times_eat=0;
	Fork leftFork, rightFork;
	public Philosopher(int id,Fork leftFork,Fork rightFork){
		this.id=id;
		this.leftFork=leftFork;
		this.rightFork=rightFork;
	}
	public void thinking(){ … }
	public void eating(){ … }
	public void run(){
		while(times_eat!=5){
			thinking();
			leftFork.up();
			rightFork.up();
			eating();
			leftFork.down();
			rightFork.down();
		}
		System.out.println("Philosopher("+id+") leaves the room");
	}
}
```

Dining.java

```
public class Dining{

	public static void main(String[] args){
		Fork f1=new Fork(1);
		Fork f2=new Fork(2);
		Fork f3=new Fork(3);
		Fork f4=new Fork(4);
		Fork f5=new Fork(5);
		new Philosopher(1,f1,f2).start();
		new Philosopher(2,f2,f3).start();
		new Philosopher(3,f3,f4).start();
		new Philosopher(4,f4,f5).start();
		new Philosopher(5,f1,f5).start();
		System.out.println("All Philosophers started");
	}
}
```

Readers – Writers Problem

from : http://greenteapress.com/semaphores/

In this problem we want to let a data structure be read and modified by concurrent threads. While the data structure is being written or modified it is often necessary to bar other threads from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data.

As in the producer-consumer problem, the solution is asymmetric. Readers and writers execute different code before entering the critical section. The synchronization constraints are:

1. Any number of readers can be in the critical section simultaneously.

2. Writers must have exclusive access to the critical section.

In other words, a writer cannot enter the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.

Readers – Writers Problem

from : http://greenteapress.com/semaphores/

A data structure can be read and modified by concurrent threads.

1. Any number of readers can be in the critical section simultaneously.

2. Writers must have exclusive access to the critical section.

In other words, a writer cannot enter the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.

River crossing problem

from : http://greenteapress.com/semaphores/

This is from a problem set written by Anthony Joseph at U.C. Berkeley.

Somewhere near Redmond, Washington there is a rowboat that is used by both Linux hackers and Microsoft employees (serfs) to cross a river. The ferry can hold exactly four people; it won't leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe.

As each thread boards the boat it should invoke a function called board. You must guarantee that all four threads from each boatload invoke board before any of the threads from the next boatload do.

After all four threads have invoked board, exactly one of them should call a function named rowBoat, indicating that that thread will take the oars. It doesn't matter which thread calls the function, as long as one does.

Don't worry about the direction of travel. Assume we are only interested in traffic going in one of the directions.

River crossing problem

from : http://greenteapress.com/semaphores/

By Anthony Joseph at U.C. Berkeley.

- There is a rowboat that is used by both **Linux hackers** and **Microsoft employees** (serfs) to cross a river.
- The ferry can hold exactly four people; it won't leave the shore with more or fewer.
- **It is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe.**
- As each thread boards the boat it should invoke a function called ***board***. You must guarantee that all four threads from each boatload invoke board before any of the threads from the next boatload do.
- After all four threads have invoked board, exactly one of them should call a function named ***rowBoat***, indicating that that thread will take the oars.

Cigarette smokers problem

from : http://greenteapress.com/semaphores/

The cigarette smokers problem problem was originally presented by Suhas Patil, who claimed that it cannot be solved with semaphores. That claim comes with some qualifications, but in any case the problem is interesting and challenging.

Four threads are involved: an agent and three smokers. The smokers loop forever, first waiting for ingredients, then making and smoking cigarettes. The ingredients are tobacco, paper, and matches.

We assume that the agent has an infinite supply of all three ingredients, and each smoker has an infinite supply of one of the ingredients; that is, one smoker has matches, another has paper, and the third has tobacco.

The agent repeatedly chooses two different ingredients at random and makes them available to the smokers. Depending on which ingredients are chosen, the smoker with the complementary ingredient should pick up both resources and proceed.

For example, if the agent puts out tobacco and paper, the smoker with the matches should pick up both ingredients, make a cigarette, and then signal the agent.

To explain the premise, the agent represents an operating system that allocates resources, and the smokers represent applications that need resources. The problem is to make sure that if resources are available that would allow one more applications to proceed, those applications should be woken up. Conversely, we want to avoid waking an application if it cannot proceed.

Patil's version imposes restrictions on the solution.Lets consider just the first, you are not allowed to modify the agent code. If the agent represents an operating system, it makes sense to assume that you don't want to modify it every time a new application comes along.

Cigarette smokers problem

from : http://greenteapress.com/semaphores/

By Suhas Patil.

- Four threads are involved: **an agent and three smokers**.
- The smokers loop forever, first waiting for ingredients, then making and smoking cigarettes.
- The ingredients are **tobacco, paper, and matches**.
- Each smoker has one of the ingredients; that is, one smoker has matches, another has paper, and the third has tobacco.
- The agent repeatedly chooses two different ingredients at random. The smoker with the complementary ingredient should pick up both resources and proceed.

Building H2O

from : http://greenteapress.com/semaphores/

This problem has been a staple of the Operating Systems class at U.C. Berkeley for at least a decade. It seems to be based on an exercise in Andrews's Concurrent Programming.

There are two kinds of threads, oxygen and hydrogen. In order to assemble these threads into water molecules, we have to create a barrier that makes each thread wait until a complete molecule is ready to proceed.

As each thread passes the barrier, it should invoke bond. You must guarantee that all the threads from one molecule invoke bond before any of the threads from the next molecule do.

In other words:

- If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.
- If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.

We don't have to worry about matching the threads up explicitly; that is, the threads do not necessarily know which other threads they are paired up with. The key is just that threads pass the barrier in complete sets; thus, if we examine the sequence of threads that invoke bond and divide them into groups of three, each group should contain one oxygen and two hydrogen threads.

Puzzle: Write synchronization code for oxygen and hydrogen molecules that enforces these constraints.

Building H2O

from : http://greenteapress.com/semaphores/

From Operating Systems class at U. California.

- There are two kinds of threads, **oxygen** and **hydrogen**.
- In order to assemble these threads into water molecules, we have to create a barrier that makes each thread wait until a complete molecule is ready to proceed.
- As each thread passes the barrier, it should invoke ***bond***. In other words:
  1. If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.
  2. If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.
- We don't have to worry about matching the threads up explicitly. The key is just that threads pass the barrier in complete sets.

The Santa Claus problem

from : http://greenteapress.com/semaphores/

This problem is from William Stallings's Operating Systems, but he attributes it to John Trono of St. Michael's College in Vermont.

Santa Claus sleeps in his shop at the North Pole and can only be awakened by either (1) all nine reindeer being back from their vacation in the South Pacific, or (2) some of the elves having difficulty making toys; to allow Santa to get some sleep, the elves can only wake him when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.) The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Here are some addition specifications:

- After the ninth reindeer arrives, Santa must invoke prepareSleigh, and then all nine reindeer must invoke getHitched.
- After the third elf arrives, Santa must invoke helpElves. Concurrently, all three elves should invoke getHelp.
- All three elves must invoke getHelp before any additional elves enter(increment the elf counter).

Santa should run in a loop so he can help many sets of elves. We can assume that there are exactly 9 reindeer, but there may be any number of elves.

Base Bibliography

- JAVA Tutorial:

  http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

- JAVA Threads, O'Reilly, chapter 4

Additional Bibliography

Multithreaded Programming, Lewis & Berg, chapter 6

Summary

Deadlock, live lock and starvation

Discussion of the solution of several classical concurrent problems

Parte II

Barreiras

Barreiras

- Uma barreira permite que um conjunto de threads se sincronize num dado ponto de código
- É inicializada com o número de threads que a usam
- Bloqueia todos os threads que a chamam até o número de threads que a usam ser zero, ponto no qual todos os threads são desbloqueados
- A ideia é estabelecer um ponto onde um conjunto de threads espera que todos os outros cheguem a esse ponto, antes de todos prosseguirem

Barreiras únicas

- A barreira única implementa um conceito similar ao de uma barreira
- A barreira única permite que um conjunto de threads (barradas) C1 espere pela chegada de um conjunto C2 de threads (expeditores). Após todos os C2 chegarem os threads do conjunto C1 avançam.
- No caso especial de uma única thread barrada:
  - cada thread expeditor incrementa a barreira única à medida que completa o seu trabalho, enquanto o thread barrado espera por todos eles
  - Uma vez que todos os threads expeditores completem o seu trabalho, o thread barrado é notificado e avança

Barreiras Únicas - Implementação

- `currentPosters, totalposters` – threads expeditores

- `currentWaiters, totalWaiters` – threads barrados

Barreiras Únicas - Implementação

```
class SingleBarrier {

  int currentPosters = 0;
  int totalPosters = 0;

  int passedWaiters = 0;
  int totalWaiters = 1;

  // ...

}
```

SingleBarrier

```
Class SingleBarrier {
   public SingleBarrier (int i) {
       totalPosters = i;
   }
   public SingleBarrier (int i, int j) {
       totalPosters = i; totalWaiters = j;
   }
   public SingleBarrier () {
   }
   public synchronized void init(int i) {
       totalPosters = i; currentPosters=0;
   }
   public synchronized void barrierSet(int i) {
       totalPosters = i; currentPosters=0;
   }
```

barrierWait

```
class SingleBarrier {
  public synchronized void barrierWait() {
    boolean interrupted = false;
    while (currentPosters != totalPosters) {
        try {wait();}
        catch (InterruptedException ie)
            {interrupted=true;}
    }
    passedWaiters++;
    if (passedWaiters == totalWaiters) {
        currentPosters = 0; passedWaiters = 0;
        notifyAll();
    }
    if (interrupted)
       Thread.currentThread().interrupt();
  }
```

Suponha que um waiter é notificado, mas um poster é mais rápido a ganhar a secção crítica...

barrierPost

```
public synchronized void barrierPost() {
  boolean interrupted = false;
  // In case a poster thread beats barrierWait,
  // keep count of posters.
  while (currentPosters == totalPosters) {
      try {wait();}
      catch (InterruptedException ie)
         {interrupted=true;}
  }
  currentPosters++;
  if (currentPosters == totalPosters) notifyAll();
  if (interrupted)
      Thread.currentThread().interrupt();
}
```

Exercício

Considere 10 threads que contam de 1 até 10^6. Inicialmente são lançados 5 threads. Quando todos esses threads chegarem a 10^3 os restantes 5 threads iniciam a contagem. Finalmente, quando todos os threads acabarem a contagem deve ser escrito no ecrã “Todos os threads acabaram de contar”.