Concurrency lecture 4

Dining Philosophers - 6.5.3

Equivalence of solutions

Problems with distributed systems - 18.2

Dining Philosophers

A philosopher thinks and eats.

5 philosophers sitting around a table.

5 forks - 1 shared between each pair of philosophers.

A philosopher needs the fork on either side in order to eat.

We don't really want philosophers starving to death.

First attempted solution with semaphores.

```
fork : array[0..4] of semaphore

procedure philosopher (name : integer); { numbered 0 to 4
}

begin
  while true do begin
    think;
    wait(fork[name]); { fork on the right }
    wait(fork[(name + 1) mod 5]); { fork on the left }
    eat;
    signal(fork[name]);
    signal(fork[(name + 1) mod 5])
  end
end;
```

What will eventually happen?

So we try to stop all philosophers getting one fork.

```
while true do begin
  think;
  simultaneous_wait(fork[name],
                    fork[(name + 1) mod 5]);
  eat;
  simultaneous_signal(fork[name],
                    fork[(name + 1) mod 5])
end;
```

The simultaneous_wait and signal operations are supposed to be atomic and block the thread until both forks are free.

No more deadlock. But the problem is still not solved.

Solutions:

- use of states for the philosophers - thinking, hungry, eating
- a hungry philosopher gets preference to over one which is thinking
- only allow 4 philosophers to pick up forks at any time
- even philosophers pick up their right forks first, odd philosophers pick up their left

Unlike the textbook which focuses on control of the philosophers I have a monitor solution which focuses on the forks. It looks very similar to the attempted simultaneous wait semaphore solution.

Why does this monitor solution work when the semaphore version didn't?

```
monitor  ForkControl;

var
  forkAvail : array[0..4] of boolean;
  forkWait : array[0..4] of conditionVariable;

procedure getBothForks(name : integer);
begin
  if not forkAvail[name] then
    wait(forkWait[name]);
  forkAvail[name] := FALSE;

  if not forkAvail[(name + 1) mod 5] then
    wait(forkWait[(name + 1) mod 5]);
  forkAvail[(name + 1) mod 5] := FALSE
end;

procedure putBackBothForks(name : integer);
begin
  forkAvail[name] := TRUE;
  forkAvail[(name + 1) mod 5] := TRUE;
  signal(forkWait[name]);
  signal(forkWait[(name + 1) mod 5]
end;
begin
  for i := 0 to 4 do
    forkAvail[i] := TRUE
end. { monitor ForkControl }
```

Each philosopher looks like this:

```
procedure philosopher(name : integer);
begin
  while TRUE do begin
    think;
    getBothForks(name);
    eat;
    putBackBothForks(name)
  end
end;
```

I tested this in Java - note changes had to be made because Java monitors don't have condition variables.

```
/ *
  ForkControl.java - monitor to keep control of the forks
  in the Dining Philosophers' problem.
  Written by Robert Sheehan.
  01/09/97
* /

public class ForkControl {

  private boolean[] forkAvailable;

  public ForkControl() {
    forkAvailable = new
boolean[DiningPhilosophers.NUMBER];
    for (int i = 0; i < DiningPhilosophers.NUMBER; i++)
      forkAvailable[i] = true;
  }
```

```
  public synchronized void getBothForks(int name) throws
InterruptedException {
    while (!forkAvailable[name])
      wait();
    forkAvailable[name] = false;

    while (!forkAvailable[(name + 1) %
DiningPhilosophers.NUMBER])
      wait();
    forkAvailable[name] =  false;
  }

  public synchronized void putBackBothForks(int name) {
    forkAvailable[name] = true;
    forkAvailable[(name + 1) %
DiningPhilosophers.NUMBER] = true;
    notifyAll(); // try it with notify()
  }

}
```

relevant philosopher code:

```
  public void run() {
    while (true) {
      think();
      eat();
    }
  }
```

```
private void think() {
    output.appendText("\ngoing to sleep");
    try {
      sleep((int)(Math.random() * 1000 + 1000));
    }
    catch (InterruptedException e) {
      output.appendText("\ninterrupted while thinking");
    }
  }

  private void eat() {
    try {
      output.appendText("\ngetting forks");
      controller.getBothForks(name);
      output.appendText("\neating");
      sleep((int)(Math.random() * 1000 + 1000));
      output.appendText("\nreturning forks");
      controller.putBackBothForks(name);
    }
    catch (InterruptedException e) {
      output.appendText("\ninterrupted while eating");
    }
  }
```

The code for the entire program is available via my lecture page on the web.

http://www.cs.auckland.ac.nz/~robert-s/415.340/lectures_1997.htm

Equivalence of solutions

Last time you wrote a semaphore object in Java, using the idea of monitors. This shows that monitors are at least as powerful as semaphores.

To show equivalence we need to show we can produce a monitor with semaphores.

A semaphore initialised to 1 is used to guard entrance to the monitor.

Wait on entry, normally signal on exit.

Condition variables complicate things

Associate a semaphore with each condition variable.

Only signal the semaphore when something is actually waiting.

Need some way of querying the semaphore queue - a common addition.

Or else keep track of this ourselves as well (no worries about mutual exclusion).

If we wake up a thread waiting on a condition variable we don't signal the entrance semaphore as we leave.

When we look at messages we see that blocking receives are equivalent to the other concurrency constructs as well.

Distributed concurrency

Locks, semaphores and monitors require shared memory.

Doesn't matter whether a single processor or multiprocessor.

Sometimes we need locks over resources which are available network wide.

No shared memory.

Which means we are going to have to send messages.

A server or coordinator process to look after the resource.

Request - Reply - Release

A process wanting the resource or mutual exclusion requests it with a message to the coordinator

and then blocks until it receives a reply.

When it receives the reply it has the resource and must send a release message when it has finished.

Everything gets more complicated because communication can be unreliable, some machines on the network might die (but others stay functional).

Coordinator may use time outs if the resource isn't released. Then it can send a query to see if the current owner is still active.

If the coordinator fails need to have an election to see which process should replace it.

The new coordinator needs to recreate a wait queue by polling all processes to see if they need the resource.

Non-centralized

Timestamps

Logical clocks

Broadcasting requests and replying