COMP30112: Concurrency

Topics 5.1: Deadlock

Howard Barringer

Room KB2.20: email: Howard.Barringer@manchester.ac.uk

March 2009
Outline

Topic 5.1: Deadlock

What is Deadlock?
Examples
Dining Philosophers
Outline

Topic 5.1: Deadlock

What is Deadlock?

Examples

Dining Philosophers
What is Deadlock

What is it?
What is Deadlock

*What is it?*
In the concurrent system:
What is Deadlock

What is it?

In the concurrent system:

- no (FSP) process has a following action
What is Deadlock

What is it?
In the concurrent system:

- no (FSP) process has a following action
- there is a (LTS) state from which no action is possible
What is Deadlock

What is it?
In the concurrent system:

- no (FSP) process has a following action
- there is a (LTS) state from which no action is possible
- all (Java) threads blocked/waiting
What is Deadlock

What is it?
In the concurrent system:

- no (FSP) process has a following action
- there is a (LTS) state from which no action is possible
- all (Java) threads blocked/waiting

Causes?
What is Deadlock

**What is it?**
In the concurrent system:
- no (FSP) process has a following action
- there is a (LTS) state from which no action is possible
- all (Java) threads blocked/waiting

**Causes?**
Four necessary and sufficient conditions:
What is Deadlock

What is it?
In the concurrent system:
- no (FSP) process has a following action
- there is a (LTS) state from which no action is possible
- all (Java) threads blocked/waiting

Causes?
Four necessary and sufficient conditions:
1. Serially reusable resources
What is Deadlock

**What is it?**
In the concurrent system:
- no (FSP) process has a following action
- there is a (LTS) state from which no action is possible
- all (Java) threads blocked/waiting

**Causes?**
Four necessary and sufficient conditions:
1. Serially reusable resources
2. Incremental acquisition
What is Deadlock

What is it?
In the concurrent system:
- no (FSP) process has a following action
- there is a (LTS) state from which no action is possible
- all (Java) threads blocked/waiting

Causes?
Four necessary and sufficient conditions:
1. Serially reusable resources
2. Incremental acquisition
3. No preemption
What is Deadlock

What is it?
In the concurrent system:

- no (FSP) process has a following action
- there is a (LTS) state from which no action is possible
- all (Java) threads blocked/waiting

Causes?
Four necessary and sufficient conditions:

1. Serially reusable resources
2. Incremental acquisition
3. No preemption
4. Wait-for cycle
Topic 5.1: Deadlock

Outline

What is Deadlock?
Examples
Dining Philosophers
FSP Example

\[ P = ( \text{starta} \rightarrow \text{startb} \rightarrow \text{stopb} \rightarrow \text{stopa} \rightarrow P ) . \]

\[ Q = ( \text{startb} \rightarrow \text{starta} \rightarrow \text{stopa} \rightarrow \text{stopb} \rightarrow Q ) . \]

\[ A = (\text{starta} \rightarrow \text{doA} \rightarrow \text{stopa} \rightarrow A) . \quad //\text{shared resource } A \]

\[ B = (\text{startb} \rightarrow \text{doB} \rightarrow \text{stopb} \rightarrow B) . \quad //\text{shared resource } B \]

\[ \text{||SYSTEM} = (a:P \ || \ b:Q \ || \ \{a,b\}::A \ || \ \{a,b\}::B) . \]
Outline

Topic 5.1: Deadlock

What is Deadlock?
Examples
Dining Philosophers
The “Dining Philosophers” Example

Classic example due to Dijkstra.

$N$ philosophers sit around a circular table
The “Dining Philosophers” Example

Classic example due to Dijkstra.

$N$ philosophers sit around a circular table
Each philosopher alternately “thinks” then “eats”
The “Dining Philosophers” Example

Classic example due to Dijkstra.

$N$ philosophers sit around a circular table
Each philosopher alternately “thinks” then “eats”
Unending supply of tangled spaghetti at table centre
The “Dining Philosophers” Example

Classic example due to Dijkstra.

\[ N \text{ philosophers sit around a circular table} \]

Each philosopher alternately “thinks” then “eats”

Unending supply of tangled spaghetti at table centre

Philosophers are poor, only 5 forks available, each one placed between two philosophers
The “Dining Philosophers” Example

Classic example due to Dijkstra.

$N$ philosophers sit around a circular table
Each philosopher alternately “thinks” then “eats”
Unending supply of tangled spaghetti at table centre
Philosophers are poor, only 5 forks available, each one placed between two philosophers
Each philosopher needs two forks to eat, those to the immediate left and right
The “Dining Philosophers” Example

Classic example due to Dijkstra.

\( N \) philosophers sit around a circular table
Each philosopher alternately “thinks” then “eats”
Unending supply of tangled spaghetti at table centre
Philosophers are poor, only 5 forks available, each one placed between two philosophers
Each philosopher needs two forks to eat, those to the immediate left and right
No philosopher must go hungry
Some Dining Philosophers
Topic 5.1: Deadlock

Some Dining Philosophers
Dining Philosophers - aspects

- concurrently executing processes
Dining Philosophers - aspects

- concurrently executing processes
- non-critical independent execution
Dining Philosophers - aspects

- concurrently executing processes
- non-critical independent execution
- simultaneous access to shared, serially reusable, resources
Dining Philosophers - aspects

- concurrently executing processes
- non-critical independent execution
- simultaneous access to shared, serially reusable, resources
- deadlock: incremental acquisition, no preemption, wait-for cycle
Dining Philosophers - aspects

- concurrently executing processes
- non-critical independent execution
- simultaneous access to shared, serially reusable, resources
- deadlock: incremental acquisition, no preemption, wait-for cycle
- several solutions
Dining Philosophers - aspects

- concurrently executing processes
- non-critical independent execution
- simultaneous access to shared, serially reusable, resources
- deadlock: incremental acquisition, no preemption, wait-for cycle
- several solutions
- fairness issues
**Topic 5.1: Deadlock**

**Dining Philosophers in FSP**

\[ \text{FORK} = (\text{get} \to \text{put} \to \text{FORK}). \]

\[ \text{PHIL} = (\text{sitdown} \to \text{right.get} \to \text{left.get} \to \text{eat} \to \text{left.put} \to \text{right.put} \to \text{getup} \to \text{PHIL}). \]

\[ \text{||DINERS}(N=5) = \]
\[ \forall [i:0..N-1] \]
\[ (\text{phil}[i] : \text{PHIL} \]
\[ \||\{\text{phil}[i].\text{left}, \text{phil}[((i-1)+N)\%N].\text{right}\}::\text{FORK}). \]
Dining Philosophers in Java - Fork

class Fork {

    private boolean taken=false;

    synchronized void put() {
        taken=false;
        notify();
    }

    synchronized void get() throws InterruptedException {
        while (taken) wait();
        taken=true;
    }
}
class Philosopher extends Thread {
    private Fork left; private Fork right;

    Philosopher(... Fork l, Fork r) {
        ... left = l; right = r; ...
    }

    public void run() {
        try {
            while (true) {
                // thinking...
                // hungry:
                right.get(); left.get();
                // eating...
                // finished eating:
                right.put(); left.put();
            } } catch (InterruptedException e) {} }
}