# Deadlock and its Detection

Shin Hong

# Deadlock Bugs Frequently Occur in Real World

| Application | What it does | Non-Deadlock | Deadlock |
|---|---|---|---|
| MySQL | Database Server | 14 | 9 |
| Apache | Web Server | 13 | 4 |
| Mozilla | Web Browser | 41 | 16 |
| OpenOffice | Office Suite | 6 | 2 |
| Total | | 74 | 31 |

- In a survey on 105 real-world concurrency bugs in open-source applications, **31 out of 105 bugs are deadlock bugs** [Lu *et al*., ASPLOS 08]

# Deadlock

- A deadlock occurs when each of a set of threads is blocked, waiting for another thread in the set to satisfy certain condition

**release shared resource**

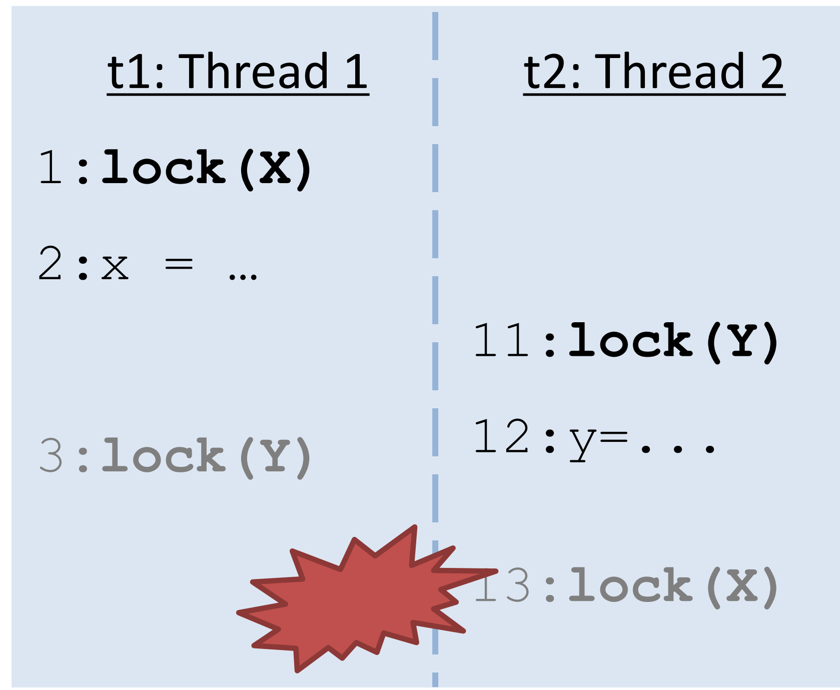
**raise event**

# Resource Deadlock in Concurrent Programs

- ABBA deadlock

```
Thread1() {
1:  lock(X)
2:  x = … ;
3:  lock(Y)
4:  y = … ;
5:  unlock(Y)
6:  unlock(X)
}
```

```
Thread2() {
11:  lock(Y)
12:  y = … ;
13:  lock(X)
14:  x = … ;
15:  unlock(X)
16:  unlock(Y)
}
```



# Communication Deadlock

- Lost notify

```
   Thread1() {
1:  ...
2:  for(i=0;i<10;i++){
3:   wait(m) ;}
   }
```

```
   Thread2() {
11:  ...
12:  for(j=0;j<10;j++){
13:   notify(m);}
   }
```

| $t_1$: Thread 1 | $t_2$: Thread 2 |
| --- | --- |
| | `13:notify(m)//j==0` |
| | … |
| | `13:notify(m)//j==1` |
| `3:wait(m)//i==0` | |
| … | |
| | … |
| | `13:notify(m)//j==9`<br>`(terminate)` |
| `3:wait(m)//i==9` | |

# Finding Deadlock Bugs is Difficult

- A deadlock bug induces deadlock situations **only under certain thread schedules**
- Systems software creates a **massive number of locks** for fine-grained concurrency controls
- **Function caller-callee relation** complicates the reasoning about possible nested lockings

# Bug Detection Approach

Resource deadlock

- Basic potential deadlock detection algorithm
- GoodLock algorithm

Communication deadlock

- CHECKMATE: a trace program model-checking technique for deadlock detection

# Basic Potential Deadlock Detection

- Extend the cyclic deadlock monitoring algorithm
- Cyclic deadlock monitoring algorithm (e.g. *LockDep*)
  - Monitor lock acquires and releases in runtime
  - Lock graph ($N$, $E_N$)
    - Create a node $n_X$ when a thread acquires lock $X$
    - Create an edge ($n_X$, $n_Y$) when a thread acquires lock $Y$ while holding lock $X$
    - Remove $n_X$ , ($n_X$, *) and (*, $n_X$) when a thread releases $X$

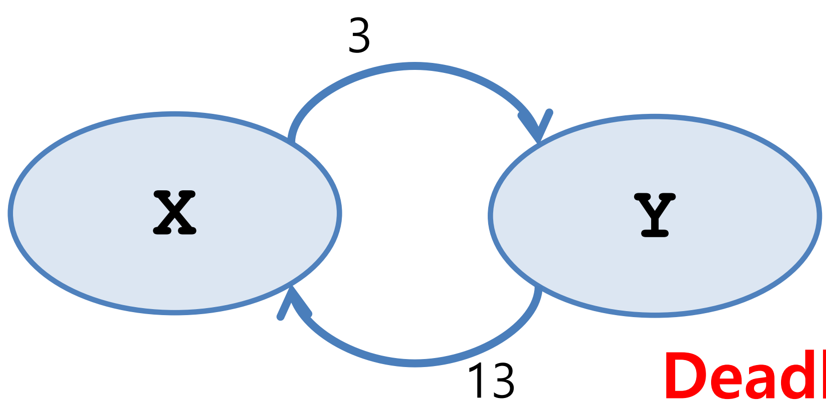
  → Report deadlock when the graph has any cycle

# Cyclic Deadlock Detection Example (1/2)

```
  Thread1() {
1:  lock(X)
2:  a = … ;
3:  lock(Y)
4:  b = … ;
5:  unlock(Y)
6:  unlock(X)
  }
```

```
  Thread2() {
11:  lock(Y)
12:  b = … ;
13:  lock(X)
14:  a = … ;
15:  unlock(X)
16:  unlock(Y)
  }
```

| t1: Thread 1 | t2: Thread 2 |
|---|---|
| `1:lock(X)` | |
| `2:a = …` | |
| | `11:lock(Y)` |
| | `12:b=...` |
| `3:lock(Y)` | |
| | `13:lock(X)` |




**Deadlock detected!**

# Cyclic Deadlock Detection Example (2/2)

```
  Thread1() {                Thread2() {
1:  lock(X);               11:  lock(Y);
2:  a = …                  12:  b = …
3:  lock(Y);               13:  lock(X);
4:  b = …                  14:  a = …
5:  unlock(Y);             15:  unlock(X);
6:  unlock(X);             16:  unlock(Y);
  }                          }
```

| t1: Thread 1 | t2: Thread 2 |
|---|---|
| 1:lock(X) | |
| 2:a = … | |
| 3:lock(Y) | |
| 4:b = … | |
| 5:unlock(Y) | |
| | 11:lock(Y) |
| 6:unlock(X) | |
| | 12:b =... |
| | 13:lock(X) |
| | 14:a =... |
| | 15:unlock(X) |
| | 16:unlock(Y) |

**No problem**

# Basic Deadlock Prediction Technique

- Potential cyclic deadlock detection algorithm [Harrow, SPIN 00]
  - Lock graph ($N$, $E_N$)
    - Create a node $n_X$ when a thread acquires lock $X$
    - Create an edge ($n_X$, $n_Y$) when a thread acquires lock $Y$ while holding lock $X$
    - ~~Remove $n_X$, ($n_X$, *) and (*, $n_X$) when a thread releases $X$~~

    → Report potential deadlocks if the resulted graph at the end of an execution has a cycle

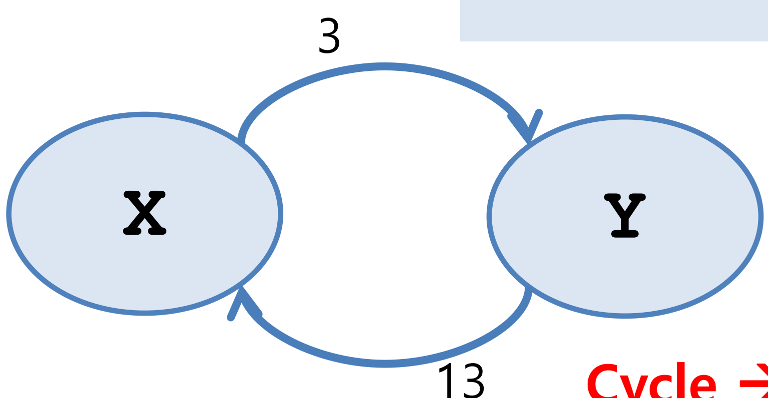
[Harrow, SPIN 00] J. J. Harrow, Jr.: Runtime checking of multithreaded applications with Visual Threads, SPIN Workshop 2000

# Potential Cyclic Deadlock Detection Example

```
Thread1() {
1:  lock(X)
2:  a = … ;
3:  lock(Y)
4:  b = … ;
5:  unlock(Y)
6:  unlock(X)
}
```

```
Thread2() {
11:  lock(Y)
12:  b = … ;
13:  lock(X)
14:  a = … ;
15:  unlock(X)
16:  unlock(Y)
}
```

| t1:Thread 1 | t2:Thread 2 |
| --- | --- |
| `1:lock(X)` | |
| `2:a = …` | |
| `3:lock(Y)` | |
| `4:b = …` | |
| `5:unlock(Y)` | |
| | `11:lock(Y)` |
| `6:unlock(X)` | |
| | `12:b=...` |
| | `13:lock(X)` |
| | `...` |




**Cycle → Potential deadlock**

# Basic Deadlock Prediction Technique

- The algorithm is commercialized as a SW tool VisualThreads (*HP*)
- Empirical results show that the algorithm is very effective to discover hidden deadlock bugs
- Challenge: generate many false positive

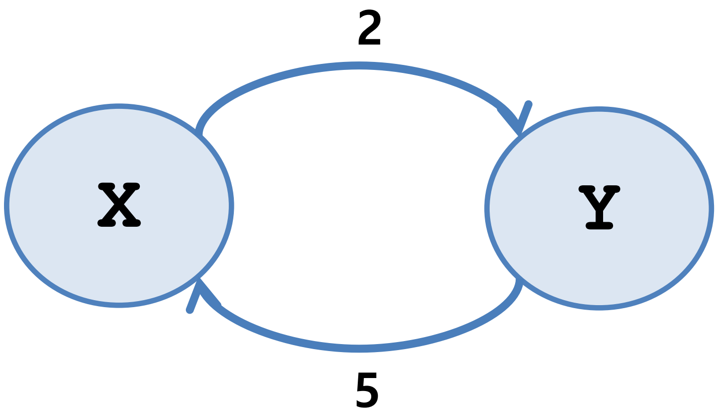
# False Positive Example#1 – Single Thread Cycle

```
  Thread1() {
1: lock(X);
2:  lock(Y);
3:  unlock(Y);
4: unlock(X);

5: lock(Y);
6:  lock(X);
7:  unlock(X);
8: unlock(Y);}
```

```
  Thread2() {
11: lock(X);
12: unlock(X);

13: lock(Y);
14: unlock(Y);}
```




The lock graph has a cycle, but no deadlock

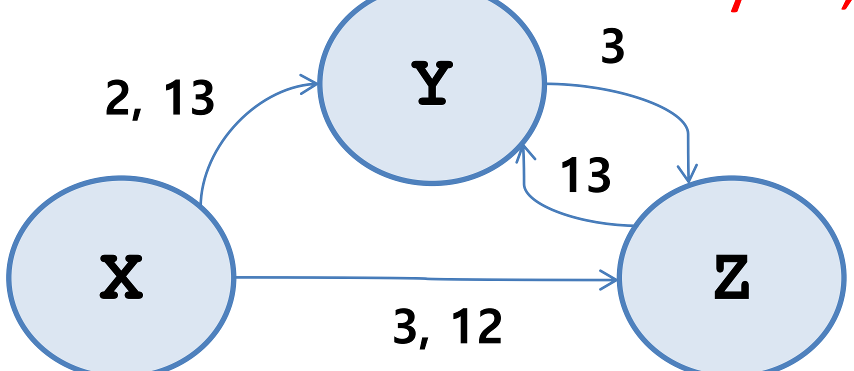
**A cycle that consists of edges created by one thread is a false positive**

# False Positive Example#2: Gate Lock

```
   Thread1() {                Thread2() {
1:  lock(X);               11:  lock(X);
2:    lock(Y);             12:    lock(Z) ;
3:      lock(Z) ;          13:      lock(Y) ;
4:      unlock(Z);         14:      unlock(Y);
5:    unlock(Y);           15:    unlock(Z);
6:  unlock(X); }           16:  unlock(X);
```

**Gate lock (guard lock)**

Cycle, but no deadlock



# False Positive Example#3: Thread Creation

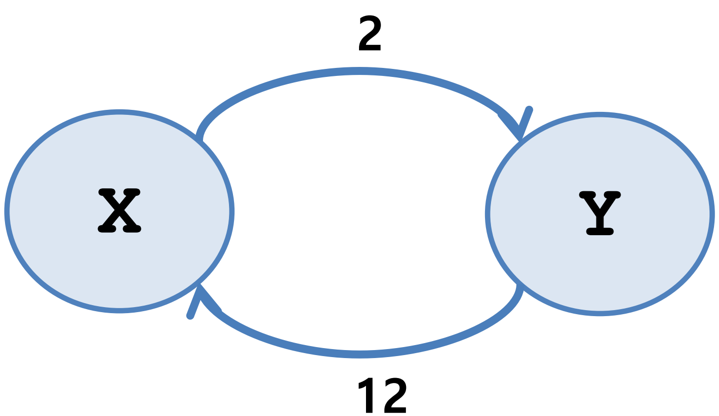
```
main(){
0:   start(f1);
}
```

```
f1(){
1:  lock(X);
2:   lock(Y);
3:   unlock(Y);
4:  unlock(X);
5:  start(f2);
}
```

Thread segment#1 (lines 1–4)

```
f2(){
11:  lock(Y) ;
12:   lock(X);
13:   unlock(X);
14:  unlock(Y);
}
```

Thread segment#2 (lines 11–14)




Cycle, but no deadlock

# GoodLock Algorithm[Agarwal, IBM 10]

- Extend the lock graph in the basic potential deadlock detection algorithm to consider *thread, gate lock, and thread segment*

- A cycle is *valid* (i.e., true positive) when every pair of edges ($m_{11}$, ($s_{11}$, $t_1$, $G_1$, $s_{12}$), $m_{12}$), and ($m_{21}$, ($s_{21}$, $t_2$, $G_2$, $s_{22}$), $m_{22}$) in the cycle satisfies:
  - $t_1 \neq t_2$, and
  - $G_1 \cap G_2 = \emptyset$ , and
  - $\neg(s_{12} \prec s_{21})$
    – The happens-before relation $\prec$ is the transitive closure of the relation $R$ such that $(s_1, s_2) \in R$ if there exists the edge from $s_1$ to $s_2$ in the thread segment graph

[Agarwal, IBM 10] R. Agarwal et al., Detection of deadlock potential in multithreaded programs, IBM Journal of Research and Development, 54(5), 2010