Chapter 6: Roadmap

6.1 Principles
6.6 Dining philosophers
6.2 Prevention
6.3 Avoidance
6.4 Detection
6.5 Integrated strategies
6.7 Unix
6.8 Linux
6.9 Solaris
6.10 Windows

Part II

Chapter 3: Processes
Chapter 4: Threads
Chapter 5: Concurrency
Chapter 6: Deadlocks
**Principles**

- Deadlock is a permanent (infinite) wait for resources
  - Requires at least two mutexes or one semaphore
- Typical example with threads P and Q:
  - Two mutexes locked in different order
  - Common source of deadlocks in more general cases
- Another example:

```java
ThreadP () {
    mutexA.Lock();
    mutexB.Lock();
    // critical section
    mutexA.Unlock();
    mutexB.Unlock();
}

ThreadQ () {
    mutexB.Lock();
    mutexA.Lock();
    // critical section
    mutexB.Unlock();
    mutexA.Unlock();
}
```

```java
CarNorth () {
    mutexA.Lock();
    mutexC.Lock();
    // drive
    mutexA.Unlock();
    mutexC.Unlock();
}

CarWest () {
    mutexC.Lock();
    mutexD.Lock();
    // drive
    mutexC.Unlock();
    mutexD.Unlock();
}
```
Example (cont’d): deadlock possible in general and...  
- Certain when each grabs their first mutex:

Conditions for a deadlock to be possible
- 1) Mutual exclusion (no sharing)
- 2) Hold and wait (allowed to hold one resource and wait for another, i.e., acquisition of multiple mutexes is *not* atomic)
- 3) No preemption (held resources not released until critical section has been successfully completed)

Conditions for it to be certain
- 1)-3) plus 4) circular wait
Assume two threads P and Q in parallel execution:
- Denote by $t$ the absolute time.
- Progress diagram is a 2D parametric curve $(x(t), y(t))$ where $x(t)$ is the number of instructions executed by Q and $y(t)$ by P.

Curves must be monotonically non-decreasing in both axes.
• Back to our example with P and Q
• Mutex places certain L-shaped obstacles/barriers on the progress diagram that cannot be crossed

ThreadP () {
  mutexA.Lock();
  mutexB.Lock();
  // critical section
  mutexA.Unlock();
  mutexB.Unlock();
}

ThreadQ () {
  mutexB.Lock();
  mutexA.Lock();
  // critical section
  mutexB.Unlock();
  mutexA.Unlock();
}
• In three quadrants near the origin, deadlock possible
  - In one, it is certain
• All other sections are safe
  - Except impossible states behind barriers

• Static or dynamic analysis to detect deadlocks
• What happens with N threads?
  - N-dimensional diagram
• How about these diagrams?
• In what order are mutexes acquired?
  - Write pseudo code for P/Q
To visualize deadlocks, often a graph is drawn between all threads and resources

- Edges of this bipartite graph are labeled with “held by” (resources → threads) and “wants” (threads → resources)

- If this directed graph has a cycle, there is a deadlock
  - Car labels (N, E, W, S) map to North/East/West/South position
Chapter 6: Roadmap

6.1 Principles
6.6 Dining philosophers
6.2 Prevention
6.3 Avoidance
6.4 Detection
6.5 Integrated strategies
6.7 Unix
6.8 Linux
6.9 Solaris
6.10 Windows
Dining Philosophers

• Yet another famous synchronization problem
  – Proposed by Dijkstra in 1965
• N philosophers are sitting at a round table with N forks between them
  – Usually N = 5 and the food is spaghetti, but this is not essential
• Each thinks for a random period of time until becoming hungry, then attempts to eat
  – Food requires usage of both adjacent forks
Dining Philosophers

- Operation of a philosopher (each is a separate thread $0 \leq i \leq N-1$)
- Forks are labeled 0 to N-1 as well

```
Mutex mutexFork[N];  // one for each fork

DropForks (int i) {
    mutexFork[i].Unlock();
    mutexFork[(i+1)%N].Unlock();
}
```

- Basic approach DPH v1.0:

```
Mutex mutexFork[N];  // one for each fork

GrabForks (int i) {
    mutexFork[i].Lock();  // right fork
    mutexFork[(i+1)%N].Lock();  // left fork
}
```

- When all are hungry, deadlock is possible
Chapter 6: Roadmap

6.1 Principles
6.6 Dining philosophers
6.2 Prevention
6.3 Avoidance
6.4 Detection
6.5 Integrated strategies
6.7 Unix
6.8 Linux
6.9 Solaris
6.10 Windows
In deadlock prevention, the algorithm is modified by programmer to make one of the 4 conditions leading to deadlock impossible.

**Condition #1: mutual exclusion**
- Typically cannot be safely eliminated (e.g., cars cannot drive on top of each other thru intersection)

**Condition #2: hold and wait**
- Can be overcome with WaitAll, DPH v1.1

```c
Mutex mutexFork[N];  // one mutex for each fork

GrabForks (int i) {
    WaitAll (mutexFork[i], mutexFork[(i+1)%N]); // both forks
}
```
- Besides speed, main drawback is that all needed mutexes must be known ahead of time and acquired in bulk.

WaitAll is either super slow (Windows) or absent (Unix)
**Condition #4: circular wait**
- Design algorithm such that a circular deadlock cannot occur

**Notice that presence of 3 or fewer cars (4 or fewer philosophers) cannot cause a cyclic wait graph**
- Use a semaphore to control how many at the table

**Q: how many can eat concurrently?**
- If only \( \lfloor N/2 \rfloor \), why allow all \( N \) to grab forks?

**How many should be allowed to use forks?**
- To achieve max concurrency, \( N-1 \), but ...

**Algorithm prone to persistent chains of waits:**

\( P_i \) (eat) \( \rightarrow \) \( P_{i+1} \) (wait) \( \rightarrow \) \( P_{i+2} \) (wait) \( \rightarrow \) \( \ldots \) \( \rightarrow \) \( P_{i+k} \) (wait)
Suppose \( T > 0 \) is the eat delay in seconds

- Max theoretical rate of algorithm is \( \frac{N}{2T} \)
- If \( T = 0 \), then mutex locking/unlocking is the bottleneck

Elegant semaphore solution, but slow

- \( T=0 \): kernel-mode semaphore kills performance
- \( T=100\text{ms} \): prone to sequential chains of waits, in which case performance may deteriorate to \( \frac{1}{T} = 10 \) per second
- Improves if think delays are random (1700/sec), or max semaphore = \( N/2 \) (1900/sec)
Another way to prevent circular wait is to request resources in the same order from all threads.

- If thread holds resource $i$ and wants $j$, then $j > i$.
  - If all other threads comply with this rule, a loop back to $i$ in the resource graph is impossible.

**DPH v1.3**

```c
CRITICAL_SECTION cs[N]; // one mutex for each fork

GrabForks (int i) {
    if (i != N-1) { // not the last guy
        EnterCriticalSection (&cs[i]);
        EnterCriticalSection (&cs[(i+1)%N]);
    } else {
        // special case, a leftie
        EnterCriticalSection (&cs[0]);
        EnterCriticalSection (&cs[N-1]);
    }
}
```

$T=0$

2M/sec $N = 5$

$T=100ms$

254/sec $N = 500$
Prevention

• **Condition #3:** no preemption of held mutexes
  – Let waiter (OS) forcefully remove forks and reassign them

• More realistic version:
  – If unable to make progress, threads can voluntarily release held mutexes, randomly sleep, and start again

• Similar to PC 3.4, which was the fastest in prior tests

```c
CRITICAL_SECTION cs[N];  // one mutex for each fork

GrabForks (int i) {
    EnterCriticalSection (&cs[i]);
    do {
        if (TryEnterCriticalSection ( &cs[ (i+1)%N ] ) != 0)
            break;
        // unable to acquire
        LeaveCriticalSection (&cs[i]);
        Sleep (rand()*DELAY);
        EnterCriticalSection (&cs[i]);
    } while (true);
}
```

<table>
<thead>
<tr>
<th>T=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.9M/sec</td>
</tr>
<tr>
<td>N = 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T=100ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>2400/sec</td>
</tr>
<tr>
<td>N = 500</td>
</tr>
</tbody>
</table>
Q: How does this program crash:

A: Deletion of invalid block from the heap
- Thrown when main() exits

Reason is that a copy of x is created to pass to Func
- This copy gets deleted when Func() returns
- Which in turn triggers destructor ~X() and deletion of buf

Finally, when main quits, it calls ~X() again
- Which attempts to delete buf a second time
• There is also a memory leak in the above scenario
• A walk-thru of what happens:

```c
main () {
    X x;
}

Func(x);
X temp;
temp = x;
Func(temp);
```

- **object x**
  - `buf = 3340`
  - `size = 100`
  - 100 bytes of RAM at address 3340

- **object temp**
  - `buf = 3490`
  - `size = 100`
  - 100 bytes of RAM at address 3490

- `calls temp’s constructor`
- `copies fields from x to temp`
- `calls Func with temp in the stack`
• Next, on return from Func(x)

  - Lesson: pass class pointers whenever feasible
    - Saves a lot of headache with copying stuff over, also faster
  - If a call-by-value is needed, use copy constructors
    - See http://en.wikipedia.org/wiki/Copy_constructor