Synchronization

Dmitri Loguinov
Texas A&M University

February 6, 2018
Chapter 5: Roadmap

5.1 Concurrency
   Appendix A.1
5.2 Hardware mutex
5.3 Semaphores
5.4 Monitors
5.5 Messages
5.6 Reader-Writer

Part II

Chapter 3: Processes
Chapter 4: Threads
Chapter 5: Concurrency
Chapter 6: Deadlocks
Inter-Process Communication (IPC)

- IPC enables exchange of information between threads/processes
- Two main approaches
  - Shared memory
  - Messages
- Shared memory
  - Primary method to pass data between threads
  - Much faster than messages
  - However, requires protection against concurrent modification to shared data
- Messages
  - Data copied through a kernel buffer
  - OS provides exclusion
  - Can be used between hosts in distributed applications (e.g., pipes, network sockets)
- Pipes already covered, now deal with shared-memory IPC
Motivation

• Most examples will be in C++ style pseudocode
  - See MSDN for detailed usage of functions
• Start with an example
  - Shared class passed to each thread
  - Thread1 computes a+b and saves into a
  - Thread2 does the same, but saves into b
• What is the outcome?

```
class Shared {
    int       a;
    int       b;
};

Shared::Thread1 ()
    a += b

Shared::Thread2 ()
    b += a

main ()
    Shared st;
    st.a = 1
    st.b = 2
    CreateThread (st.Thread1)
    CreateThread (st.Thread2)
    print (st.a, st.b)
```

• Prints (1,2) and quits
  - Need to wait for threads
  - Assuming this problem is fixed, what is the result?
Motivation

- Analyze the various execution paths
  - Two threads concurrently execute this:

```plaintext
thread 1
Shared::Thread1 ()
  1) a += b

thread 2
Shared::Thread2 ()
  2) b += a
```

- CPU trace:

```plaintext
ver 1
  1) a = 3, b = 2
  2) a = 3, b = 5
  main prints (3,5)

ver 2
  1) a = 4, b = 3
  2) a = 1, b = 3
  main prints (4,3)

ver 3
  1) reads a,b into registers
  2) reads a,b into registers
  1) computes sum, saves a = 3
  2) computes sum, saves b = 3
  main prints (3,3)
```

- non-deterministic result that depends on who gets there first (race condition)
- unintended result (depends on compiler)

// initial state
st.a = 1
st.b = 2
Motivation

• How about the next example
  – Now both variables are modified, threads print their values

thread 1

Shared::Thread1 ()
1)    a += b
2)    b += a
3)    print (a, b)

thread 2

Shared::Thread2 ()
4)    a = 2*a + b
5)    b = a + 2*b
6)    print (a, b)

• CPU trace:

<table>
<thead>
<tr>
<th></th>
<th>ver 1</th>
<th>ver 2</th>
<th>ver 3</th>
<th>ver 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>a = 3, b = 2</td>
<td>a = 3, b = 2</td>
<td>a = 3, b = 2</td>
<td>a = 3, b = 2</td>
</tr>
<tr>
<td>2)</td>
<td>a = 3, b = 5</td>
<td>a = 8, b = 2</td>
<td>a = 3, b = 5</td>
<td>a = 8, b = 2</td>
</tr>
<tr>
<td>3)</td>
<td>prints (3,5)</td>
<td>2) a = 8, b = 10</td>
<td>4) a = 11, b = 5</td>
<td>2) a = 8, b = 10</td>
</tr>
<tr>
<td>4)</td>
<td>a = 11, b = 5</td>
<td>5) a = 8, b = 28</td>
<td>5) a = 11, b = 21</td>
<td>3) prints (8,10)</td>
</tr>
<tr>
<td>5)</td>
<td>a = 11, b = 21</td>
<td>3) prints (8,28)</td>
<td>3) prints (11,21)</td>
<td>5) a = 8, b = 28</td>
</tr>
<tr>
<td>6)</td>
<td>prints (11,21)</td>
<td>6) prints (8,28)</td>
<td>6) prints (11,21)</td>
<td>6) prints (8,28)</td>
</tr>
</tbody>
</table>
Motivation

- Example (cont’d)
  - How many possible execution traces?
  - Build an execution tree:

Generalization: for two threads with $m$ and $n$ instructions respectively, the number of possible ways to interleave them:

$$\binom{m + n}{m}$$

For $m = n = 100$, this is $10^{59}$

symmetric subtree omitted
Motivation

- Actual tree is deeper since we have to consider each assembler-level instruction
  - Even most basic $c = a + b$ may be implemented as 4 CPU instructions: load (reg1, a), load(reg2, b), add(reg1, reg2), store (c, reg1)
  - Also could be load(reg, a), add(reg, b), store (c,reg)
- Because of this, synchronization bugs may be compiler-specific
  - Some may only appear in debug or release mode
- Conclusion: proper synchronization is mandatory for access to shared memory
- However, not all access needs protection
  - Required only if data is modified by at least one thread
Terminology

- **Critical section**
  - Piece of code that is sensitive to concurrent events in other threads

- **Critical sections require synchronization to exclude other threads from damaging data**

- **Atomic operation**
  - Set of instructions that cannot be interrupted by another thread

- **Single CPU instruction is always atomic**
  - Is the code above safe?

- **Nope, L2/L3 cache coherency problems on multi-core platforms**
  - Result unpredictable

- **Also, compiler may split this into multiple instructions**
  - Possible in debug mode

- **Deadlock**
  - Infinite wait for events or some conditions
Deadlock Illustrated
**Terminology**

- **Liveloak**
  - Non-stop activity that typically changes *shared state*, but makes no progress
  - Unlike deadlock, which makes no change to shared variables

- **Elevator example:**
  - Every time a button is pressed, elevator responds by moving towards the floor where it was pressed
  - New button commands preempt old ones
  - Selfish customers
Terminology

- Mutual exclusion (mutex)
  - Condition under which only one thread can be in its critical section at one time
- Multiple critical sections within a thread possible
- Race condition
  - Situation where the outcome depends on the order of thread execution
  - Hw1-part3: robots race to find the exit; found solution is non-deterministic
  - Sometimes acceptable

- Busy-spinning
  - A while loop that tests variable(s) until some condition is reached
  - Must be used very carefully to avoid locking up the CPU

- Work starvation
  - Certain threads are indefinitely prevented from performing work

Shared::Thread()
  MutexA.Lock() // enter
  a++
  MutexA.Unlock() // leave
  // do some work here
  MutexB.Lock() // enter
  b++
  c += b
  MutexB.Unlock() // leave
Terminology

• Work starvation (cont’d)
  - Caused by other threads stealing all the work or OS scheduler never allowing certain threads to run
• Assuming the OS is well-designed, only the former issue is of concern
• Example
  - Hw1-part3: one thread deposits new rooms in the queue, then immediately grabs them all back

• What does this code do if pipe is closed by CC:

```c
while (exit not found)
    DWORD read = 0;
    ReadFile (pipe, buf, allocatedSize, &read);
    // deal with overflow, read rooms
```

  - Misses rooms

• Are concurrent threads safe running this loop:

```c
while (exit not found)
    DWORD read = 0;
    ReadFile (pipe, buf, allocatedSize, &read);
    // deal with overflow, read rooms
```

  - No, need a mutex