CSCE 313-200
Introduction to Computer Systems
Spring 2024

Synchronization

Dmitri Loguinov
Texas A&M University

February 2, 2024

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 two threads
  - 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

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

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

thread 1

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

thread 2

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

- CPU trace:

ver 1

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

ver 2

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

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

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)
```

unintended result (depends on compiler)

Motivation

- How about the next example
  - Now both variables are modifed, 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:

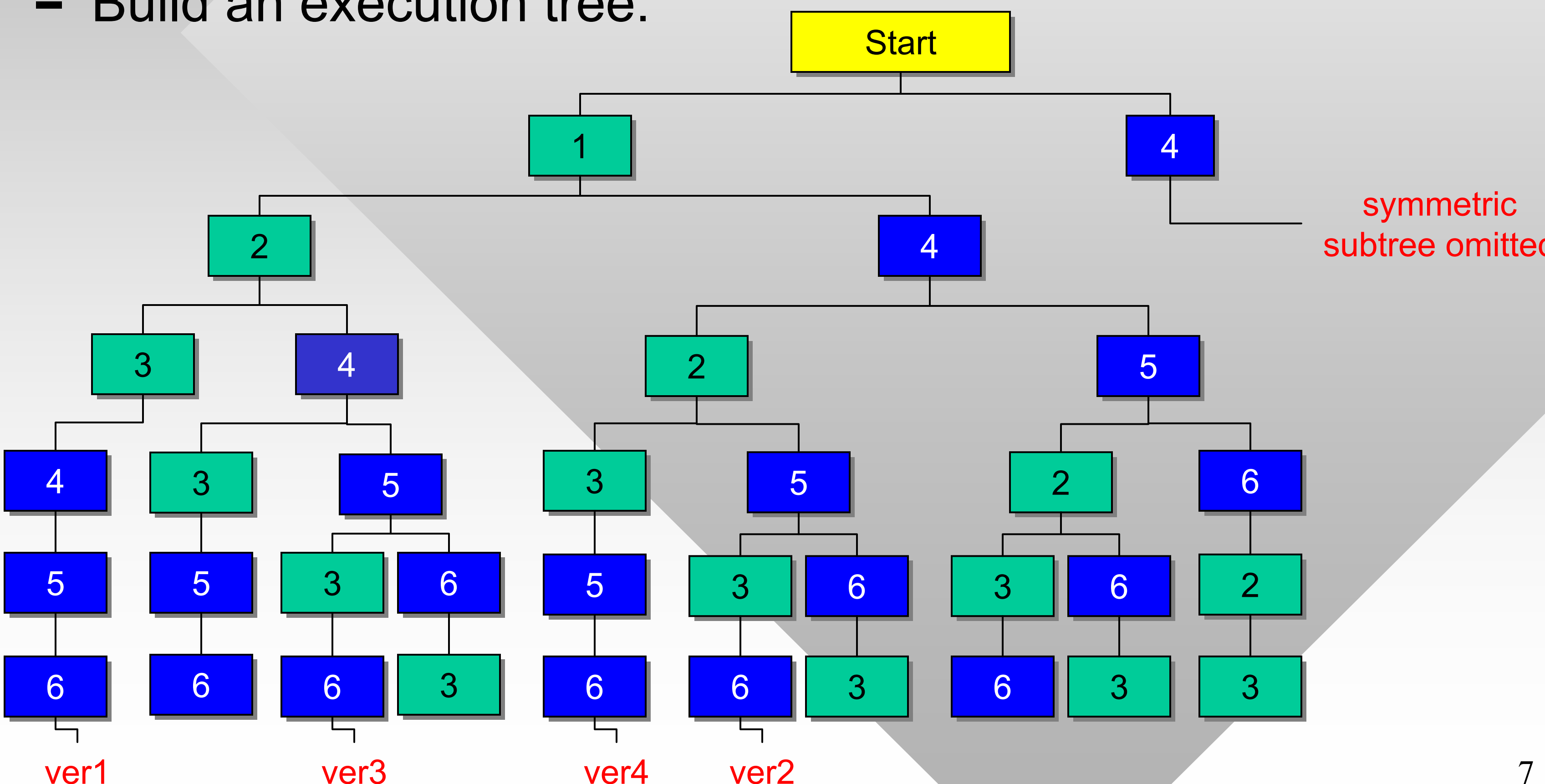
ver 1	ver 2	ver 3	ver 4
1) a = 3, b = 2	1) a = 3, b = 2	1) a = 3, b = 2	1) a = 3, b = 2
2) a = 3, b = 5	4) a = 8, b = 2	2) a = 3, b = 5	4) a = 8, b = 2
3) prints (3,5)	2) a = 8, b = 10	4) a = 11, b = 5	2) a = 8, b = 10
4) a = 11, b = 5	5) a = 8, b = 28	5) a = 11, b = 21	3) prints (8,10)
5) a = 11, b = 21	3) prints (8,28)	3) prints (11,21)	5) a = 8, b = 28
6) prints (11,21)	6) prints (8,28)	6) prints (11,21)	6) prints (8,28)

Motivation

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}

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



Motivation

- Actual tree is deeper since we have to consider each assembly-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

```
Shared::Thread ()
    a++
```

- 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

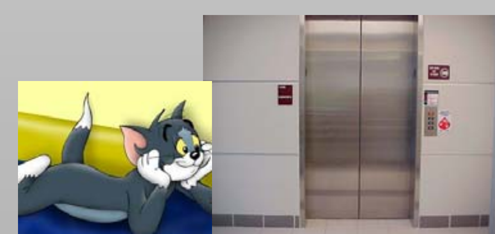
Deadlock Illustrated

Terminology

- Livelock
  - 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


floor 10


floor 5


floor 1

Terminology

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

- Mutual exclusion (mutex)
  - Data structure that allows only one thread 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
  - Not used often in user space, but parts of the kernel rely on it
- Work starvation
  - Certain threads are under-utilized (ready to run, but no work)

Terminology

- Work starvation (cont'd)
  - Caused by unbalanced job partitioning or OS scheduler giving less CPU time to certain threads
- Assuming the OS is well-designed, only the former issue is of concern
- Examples
  - Hw1-part3: one thread deposits new rooms in the queue, then immediately grabs them all back for exploration
  - Threads sort keys concurrently, where thread i gets keys whose upper k bits are i
- Does this search loop require a mutex:

```
while (exit not found)
    x = U.pop();
    Expore(x);
```

  - Yes since U.pop() modifies the underlying data structure
- Should Explore(x) be inside a mutex?