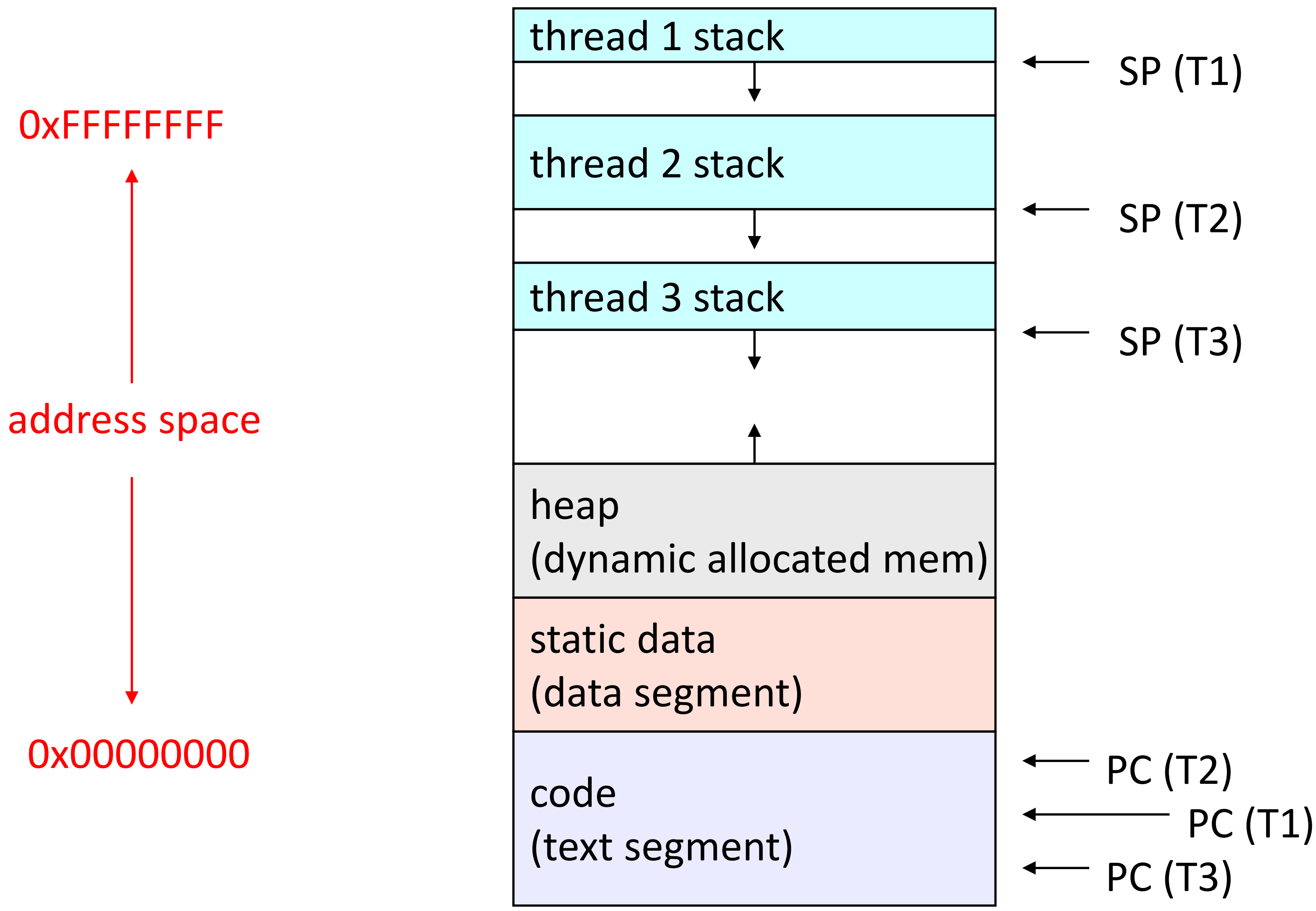
Computer Systems

CSE 410 Winter 2022

17 –Race Conditions / Critical Sections / Mutual Exclusion

Address space with threads



Race Conditions

- A race condition is code whose result may depend on the timing of the threads' executions
  - Result obtained depends on things that can't be predicted, like dynamic decisions of the scheduler or cache hits/misses

Example Race Condition

```
Starting worker threads
Thread A here
Thread B here
Thread B here
Thread B here
Thread A here
Thread A here
Thread B here
Thread A here
Thread B here
Thread B here
Thread A here
Thread B here
Thread A here
Thread B here
Thread A here
Thread B here
Thread A here
Thread A here
Thread B here
Thread A here
Worker threads have terminated
```

```
Starting worker threads
Thread A here
Thread B here
Thread A here
Thread A here
Thread B here
Thread A here
Thread A here
Thread B here
Thread B here
Thread B here
Thread A here
Thread B here
Thread A here
Thread B here
Thread B here
Thread A here
Thread A here
Thread B here
Thread B here
Thread A here
Worker threads have terminated
```

Race Conditions

- Race conditions are generally undesirable
  - They're undesirable unless every interleaving of execution of statements by the threads results in an outcome that is considered correct

Critical Sections

- Critical sections are sections of code that may not get the right result if executed by more than one thread at a time (but will get the right result if executed by only one thread at a time)
  - They're a particular kind of race condition

- An example: x = x + 1
  - If x is 0 and two threads execute this statement at once (on the same variable x), the final result may be 1 or it may be 2

- Why?

Critical Sections: read-modify-write

- x = x + 1 generates assembler like this:
  - ```
    lw    x6, x(x0)
    addi  x6, x6, 1
    sw    x6, x(x0)
    ```
- Assume x starts out with value 0 and two threads execute this code at different times

Cycle	x6 on core 0 (thread 0)	x6 on core 3 (thread 1)
0	lw: 0	--
1	addi: 1	--
2	sw: 1	--
3	---	lw: 1
4	---	addi: 2
5	---	sw: 2
6	---	
    ```

Critical Sections: read-modify-write

- x = x + 1 generates assembler like this:
  - ```
    lw    x6, x(x0)
    addi  x6, x6, 1
    sw    x6, x(x0)
    ```
- Assume x starts out with value 0 and two threads execute this code concurrently

Cycle	x6 on core 0 (thread 0)	x6 on core 3 (thread 1)
0	lw: 0	--
1	addi: 1	lw: 0
2	sw: 1	addi:1
3	---	sw: 1
4	---	--
    ```

Critical Sections

- Critical sections happen when two or more threads apply a read-modify-write operation to the same memory (variable)
  - Fetch a value
  - Compute a new value based on the fetched value
  - Write the new value back to memory
- Critical sections do not happen when
  - Threads are just reading
  - Threads are operating on different sets of variables (e.g., locals)

Synchronization: mutual exclusion

- Critical sections are resolved by ensuring that at most one thread executes the code within them at a time
- That kind of synchronization is called *mutual exclusion*

- One way to achieve mutual exclusion is through *synchronization variables*

Synchronization Variables: Locks

- Locks (sometimes called mutexes) are synchronization variables with two states:
  - locked
  - unlocked
- and two operations
  - lock()
  - unlock()

- A thread calling lock()
  - changes the lock state to lock, if it is currently unlocked
  - blocks, if the lock is currently locked

- A thread calling unlock()
  - if there are no threads blocked on the lock, changes the lock state to unlocked()
  - if there are threads blocked on the lock, unblocks one of them

Locks and Critical Sections

- Example:

```
lock  incrementLock;
int x = 0;
...
// critical section code possibly executed by multiple threads
incrementLock.lock();
x = x + 1;
incrementLock.unlock():
```

- There can be only one thread performing x=x+1 at a time, so the value of x at all times equals the number of times that line of code has been executed by any thread

Other Examples of Critical Section

- Inserting an element into a data structure (e.g., linked list, binary search tree)
- Removing an element from a data structure
- Allocating space in the heap
- Creating a new thread (e.g., stack allocation)

Problems with Locks

- lock() and unlock() are somewhat expensive
  - Can be very noticeable if the amount of work in the critical section is small
- buggy code might have a code path that fails to unlock() a lock
  - Eventually, the application seems to just hang
    - Solution: (A) Debug!
      (B) Use language support to enforce unlocking
      For instance:

```
lock(myLock) {
        ... critical section code
}
```

  - Code with more than one lock might deadlock

```
lock(lockA)
...
lock(lockB)
...
unlock(lockB)
...
unlock(lockA)
```

```
lock(lockB)
...
lock(lockA)
...
unlock(lockA)
...
unlock(lockB)
```

Solution: always acquire locks in a particular order

Java: Synchronizing Access to Objects

- The Java programming language supports threads and thread synchronization as part of the language
- We've seen the Thread class...

- Synchronization:
  - Java provides support for mutual exclusion of operations on an object
  - All classes are subclasses of Object
  - An Object has a lock
  - A method can be annotated as "synchronized"
  - Only one synchronized method can be executed at a time
    - Entering the synchronized method requires locking the object lock
    - Leaving the synchronized method unlocks the object lock

Java Example (klaatu:/courses/cse410/22wi/Java-race-example/)

- N worker threads increment a shared counter K times and then decrement it K times
  - x = x + 1 // K times
  - x = x – 1 // K times
    s
- If the workers are correctly synchronized, the final value will be 0

Worker / (Unsynchronized) Counter Classes

```
public class Worker extends Thread {
   Counter  c;
   int      iterations;

   Worker(Counter c, int iterations) {
      this.c = c;
      this.iterations = iterations;
   }

   public void run() {
      // count up
      for (int i=0; i<iterations; i++) {
         c.increment(1);
      }
      // count down
      for (int i=0; i<iterations; i++) {
         c.increment(-1);
      }
   }
}
```

```
public class Counter {

   protected int count = 0;

   public void increment(int amount) {
      count += amount;
   }

   public int getValue() {
      return count;
   }
}
```

Unsynchronized Counter Results: Incorrectness

```
$ java Unsynch 1000000 1
Count = 0
$ java Unsynch 1000000 2
Count = 96097
$ java Unsynch 1000000 3
Count = 16789
$ java Unsynch 1000000 3
Count = 264651
$ java Unsynch 1000000 4
Count = -198217
```

Worker / (Synchronized) Counter Classes

```
public class Worker extends Thread {
   Counter  c;
   int      iterations;

   Worker(Counter c, int iterations) {
      this.c = c;
      this.iterations = iterations;
   }

   public void run() {
      // count up
      for (int i=0; i<iterations; i++) {
         // spend some time
         for (int j=0; i<100; i++) {
         }
         c.increment(1);
      }
      // count down
      for (int i=0; i<iterations; i++) {
         for (int j=0; i<100; i++) {
         }
         c.increment(-1);
      }
   }
}
```

```
public class SynchronizedCounter extends Counter {
    protected int count=0;

   public synchronized void increment( int amount) {
      count += amount;
   }

   public int getValue() {
      return count;
   }
}
```

Synchronized Counter Results: Correctness

```
$ java Synch 1000000 1
Count = 0
$ java Synch 1000000 2
Count = 0
$ java Synch 1000000 3
Count = 0
$ java Synch 1000000 3
Count = 0
$ java Synch 1000000 4
Count = 0
```

What About Performance?: Unsynchronized

Arguments are:
(1) value to count to
(2) number of threads to use

The (constant) total number of iterations is divided as equally as possible among the threads.

```
$ time java Unsynch 100000000 1
Count = 0
real    0m0.898s

$ time java Unsynch 100000000 2
Count = -6938115
real    0m0.744s

$ time java Unsynch 100000000 3
Count = -1669864
real    0m0.620s

$ time java Unsynch 100000000 4
Count = 1321204
real    0m0.531s
```

What About Performance?: Synchronized

$ time java Synch 100000000 1
Count = 0
real 0m1.201s

vs. 0.898s unsynchronized

$ time java Synch 100000000 2
Count = 0
real 0m19.623s

16 times slower!

$ time java Synch 100000000 3
Count = 0
real 0m20.131s

Pretty much the same slower

Performance is complicated...

Speedup: A Measure of Parallel Performance

- Speedup S(p) is defined as
  (execution time with 1 core) / (execution time with p cores)

- Speedup is generally sub-linear
  - You won't get a speedup of 6 running on six cores, you'll get something less

- Why?

Amdahl’s Law

- Let **F** be the fraction of the execution time on 1 core that is “inherently sequential”
  - Cannot be speeded up

- Then Speedup(infinity) <= 1/**F**

- Example: if 10% of the single core computation time is inherently sequential, speedup due to parallel execution can never be greater than 10

- “Proof”: S(p) <= 1 / (**F** + (1-**F**)/p)

Summary

- Multiple threads within one process are useful to:
  - Simplify the code – multiple logically distinct control flows are easier to express that way than as one control flow that hops around from one activity to another
    - The current homework
  - Employ parallelism (the simultaneous use of multiple cores) to try to improve performance

- When you have multiple threads you have to worry about coordinating the order of operations performed by each
  - Race conditions – unpredictable results
  - Critical sections – possibly incorrect results

Summary (cont.)

- We order the computations performed by different threads using synchronization variables
  - Locks – only one thread may hold the lock at any time
    directly relevant to achieving mutual exclusion for critical sections

- Understanding the performance you'll achieve using parallelism isn't straightforward
  - There are overheads imposed (relative to single threaded) by the need to synchronize the threads
  - There are memory interactions – data may have to move from one core to another, which is slow

- Amdahl's Law gives an upper bound on potential parallel performance, showing it is limited by the fraction of the work that is "inherently sequential"