

CSC 533: Programming Languages

Spring 2016

Concurrency

- levels of concurrency
- synchronization
 - competition vs. cooperation, race conditions
- synchronization mechanisms
 - semaphores, monitors, message passing
- concurrency in Java
 - threads, Thread/Runnable, synchronize

1

Concurrency

many modern computers utilize multiple processors or multicores

- can speed up processing by executing statements/programs concurrently
- e.g., a Web server may have multiple processors to handle requests
- e.g., a Web browser might download/request/render concurrently

concurrency can occur at four levels

- machine instruction level – handled at hardware level
- *high-level language statement level*
- *unit (subprogram) level*
- program level – handled at OS level

concurrency can be

- *physical* - multiple independent processors (multiple threads of control)
- *logical* – even with one processor, can time-share to (e.g., browser tasks)

2

Subprogram-level concurrency

a *task* or *process* or *thread* is a program unit that can be in concurrent execution with other program units

tasks differ from ordinary subprograms in that:

- a task may be implicitly started
- when a program unit starts the execution of a task, it is not necessarily suspended
- when a task's execution is completed, control may not return to the caller

tasks can be:

- *heavyweight* – each executes in its own address space
- *lightweight* – all tasks execute in the same address space (more efficient)

since tasks are rarely disjoint, there must be mechanisms for coordinating or *synchronizing* tasks

3

Cooperation synchronization

sometimes, tasks must cooperate

- task A must wait for task B to complete some activity before it can begin or continue its execution
- e.g., producer/consumer relationship

task A constantly monitors the keyboard, reading each new input and storing in a buffer

– can't store if the buffer is full

task B constantly monitors the buffer, accesses each new input and removes from the buffer

– can't access if the buffer is empty

4

Competition synchronization

sometimes tasks compete for resources

- tasks A and B both require access to a non-shareable resource – must prevent simultaneous access to preserve integrity
- e.g., suppose tasks A & B access a shared variable `TOTAL`

A executes `TOTAL += 1;` B executes `TOTAL *= 2;`

at the machine-language level, each assignment involves 3 steps:

- fetch the value of `TOTAL`
- perform the operation
- store the result in `TOTAL`

5

Race condition

if the shared variable can be accessed "simultaneously," the interleaving of steps can produce different results

- B fetches `TOTAL` (3)
- A fetches `TOTAL` (3)
- B performs operation (6)
- A performs operation (4)
- B stores `TOTAL` (6)
- A stores `TOTAL` (4)

- B fetches `TOTAL` (3)
- B performs operation (6)
- B stores `TOTAL` (6)
- A fetches `TOTAL` (6)
- A performs operation (7)
- A stores `TOTAL` (7)

are other results possible?

- A fetches `TOTAL` (3)
- B fetches `TOTAL` (3)
- A performs operation (4)
- B performs operation (6)
- A stores `TOTAL` (4)
- B stores `TOTAL` (6)

- A fetches `TOTAL` (3)
- A performs operation (4)
- A stores `TOTAL` (4)
- B fetches `TOTAL` (4)
- B performs operation (8)
- B stores `TOTAL` (8)

6

Synchronization mechanisms

3 methods for providing mutually exclusive access to a resource

1. semaphores
 - early, simple approach (Dijkstra, 1965)
2. monitors
 - incorporates data abstraction (Brinch Hansen, 1973)
3. message passing
 - utilizes rendezvous messages (Brinch Hansen and Hoare, 1978)
(see text for examples - will not discuss here)

7

Semaphores

a *semaphore* is a data structure consisting of a counter and a queue for storing task descriptors

- counter represents a number of available resources (initially some N)
- queue represents tasks waiting for resources (initially empty)
- *wait operation*: allocate resource if available, else enqueue the task
- *release*: deallocate the resource, reassign to a task if waiting

```
wait (aSemaphore)
if aSemaphore's counter > 0 then
  decrement aSemaphore's counter
else
  put the caller in aSemaphore's queue
  attempt to transfer control to some ready task
  (if the task ready queue is empty, deadlock occurs)
end if
```

```
release (aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
  increment aSemaphore's counter
else
  put the calling task in the task-ready queue
  transfer control to a task from aSemaphore's queue
end
```

- can be used for both competition and cooperation synchronization

8

Semaphores for competition synchronization

```
semaphore access;
access.count = 1;

task A;
...
wait(access);    {wait for access}
TOTAL += 1;
release(access); {relinquish access}
...
end A;

task B;
...
wait(access);    {wait for access}
TOTAL *= 2;
release(access); {relinquish access}
...
end B;
```

wait & release must be implemented as single machine instructions
WHY?

9

Semaphores for cooperation synchronization

```
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLen;

task producer;
loop
-- produce VALUE --
wait (emptyspots); {wait for space}
DEPOSIT(VALUE);
release(fullspots); {increase filled}
end loop;
end producer;

task consumer;
loop
wait (fullspots);{wait till not empty}
FETCH(VALUE);
release(emptyspots); {increase empty}
-- consume VALUE --
end loop;
end consumer;
```

java.util.concurrent.Semaphore
defines a Semaphore class with
acquire & release methods

10

Evaluation of semaphores

assuming they are indivisible, wait & release can be used to provide competition and cooperation synchronization

however, the programmer must use them correctly

- *forgetting to wait can lead to mutual access*
- *forgetting to release can lead to deadlock (infinite waiting)*

- *for producer/consumer, can lead to buffer overflow or underflow*

"The semaphore is an elegant synchronization tool for an ideal programmer who never makes mistakes." (Brinch Hansen, 1978)

11

Monitors

a monitor is an abstract data type that encapsulate the shared data and its operations

- originally implemented in Concurrent Pascal (1975)
- supported by Java, Ada, C#, ...

since the shared data is resident in the monitor (rather than in the client code), monitor operations can control access

- monitor implementation guarantees synchronized access by allowing only one access at a time
- calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call

Java automatically associates a monitor with each object

- can force mutual exclusion by declaring methods to be `synchronized`

12

Example: producer/consumer in Java

Java supports concurrency via threads (lightweight processes)

- `public static void main` automatically spawns a thread
- users can create additional threads by extending the `Thread` class (or by implementing the `Runnable` interface)
- a `Thread` class must override `run()`, which specifies the action of that thread

```
class Producer extends Thread {
    private Buffer buffer;

    Producer(Buffer b) {
        this.buffer = b;
    }

    public void run() {
        for (int t = 1; t <= 10; t++) {
            System.out.println("Produced task " + t);
            this.buffer.put(t);
        }
    }
}
```

```
class Consumer extends Thread {
    private Buffer buffer;

    Consumer(Buffer b) {
        this.buffer = b;
    }

    public void run() {
        for (int i = 1; i <= 10; i++) {
            int t = this.buffer.get();
            System.out.println("Consuming task " + t);
        }
    }
}
```

13

```
public class Buffer {
    private int [] buffer;
    private int numStored, putIndex, getIndex;

    public Buffer(int size) {
        this.buffer = new int[size];
    }

    public synchronized void put(int task) {
        while(this.numStored == this.buffer.length) {
            try { wait(); }
            catch (InterruptedException e) { }
        }
        this.buffer[this.putIndex] = task;
        this.putIndex =
            (this.putIndex + 1) % this.buffer.length;
        this.numStored++;
        notify();
    }

    public synchronized int get() {
        while (this.numStored == 0) {
            try { wait(); }
            catch (InterruptedException e) { }
        }
        int task = this.buffer[this.getIndex];
        this.getIndex =
            (this.getIndex + 1) % this.buffer.length;
        this.numStored--;
        notify();
        return task;
    }
}
```

here, Buffer contains an array of ints

- will treat as a circular queue
- `putIndex` will keep track of next place to put a value (wraps around)
- `getIndex` will keep track of next place to get a value (wraps around)
- `numStored` keeps track of number currently stored

useful Thread methods

- `start()` spawns the thread (i.e., calls its `run` method)
- `wait()` suspends the current thread (and releases the monitor)
- `notify()` wakes up a thread waiting for the monitor

```
public static void main(String [] args) {
    Buffer b = new Buffer(4);
    Producer p = new Producer(b);
    Consumer c = new Consumer(b);

    p.start();
    c.start();
}
```

14

Example: summing an array

consider the task of summing all of the numbers in an array

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
nums	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98

```
public static int sum(int[] nums) {  
    int total = 0;  
    for (int i = 0; i < nums.length; i++) {  
        total += nums[i];  
    }  
    return total;  
}
```

- brute force algorithm is $O(N)$, so doubling the size doubles the time

15

Example: summing an array

if we had 2 CPUs/cores, could sum each half separately, then combine

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
nums	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98

$$\begin{aligned} \text{sum1} &= 22+18+12+-4+27+30+36+50 = \mathbf{191} & \text{sum2} &= 7+68+91+56+2+85+42+98 = \mathbf{449} \\ \text{sum} &= \text{sum1} + \text{sum2} = \mathbf{640} \end{aligned}$$

- note: still $O(N)$, but reduces the constant **BY HOW MUCH?**

16

Example: summing an array

```
public class SumThread extends Thread {
    private int[] nums;
    private int minIndex;
    private int maxIndex;
    private int computedSum;

    public SumThread(int[] nums, int minIndex, int maxIndex) {
        this.nums = nums;
        this.minIndex = minIndex;
        this.maxIndex = maxIndex;
        this.computedSum = 0;
    }

    public int getSum() {
        return this.computedSum;
    }

    public void run() {
        this.computedSum = 0;
        for (int i = this.minIndex; i < this.maxIndex; i++) {
            this.computedSum += this.nums[i];
        }
    }
}
```

note: the `run` method does not have any parameters

- must store needed values in fields when construct the thread
- can have other methods as well

17

Example: summing an array

```
public class ArraySum {

    public static int sumConcurrently(int[] a, int threadCount) {
        int len = (int) Math.ceil(1.0 * a.length / threadCount);
        Thread[] threads = new Thread[threadCount];
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new SumThread(a, i*len, Math.min((i+1)*len, a.length));
            threads[i].start();
        }
        try {
            for (Thread t : threads) {
                t.join();
            }
        } catch (InterruptedException ie) {}

        int total = 0;
        for (Thread summer : threads) {
            total += ((SumThread) summer).getSum();
        }
        return total;
    }

    . . .
}
```

here, the array is divided into `threadCount` segments

- each spawns a `SumThread` to process

the `join` method coordinates by forcing the program to wait for each thread to finish

18

Example: summing an array

```
public static void main(String[] args) {
    Random randy = new Random();
    int size = 1000;

    System.out.println("Enter number of threads: ");
    Scanner input = new Scanner(System.in);
    int numThreads = input.nextInt();
    input.close();

    while (true) {
        int[] nums = new int[size];
        for (int j = 0; j < size; j++) {
            nums[j] = randy.nextInt();
        }

        long start = System.currentTimeMillis();
        int total = 0;
        for (int j = 1; j <= 1000; j++) {
            total = sumConcurrently(nums, numThreads);
        }
        long end = System.currentTimeMillis();

        System.out.printf("%10d elements => %6d microsec\n", size, end-start);
        size *= 2;
    }
}
```

driver program prompts for
the number of threads

since timings are so fast,
actually performs 1000
sums

19

```
Enter number of threads:
1
 1000 elements => 89 microsec
 2000 elements => 91 microsec
 4000 elements => 95 microsec
 8000 elements => 91 microsec
16000 elements => 109 microsec
32000 elements => 107 microsec
64000 elements => 118 microsec
128000 elements => 137 microsec
256000 elements => 184 microsec
512000 elements => 256 microsec
1024000 elements => 402 microsec
2048000 elements => 858 microsec
4096000 elements => 1691 microsec
8192000 elements => 3125 microsec
16384000 elements => 6062 microsec
32768000 elements => 11836 microsec
65536000 elements => 22726 microsec
131072000 elements => 45429 microsec
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at ArraySum.main(ArraySum.java:52)
```

```
Enter number of threads:
2
 1000 elements => 143 microsec
 2000 elements => 151 microsec
 4000 elements => 154 microsec
 8000 elements => 168 microsec
16000 elements => 162 microsec
32000 elements => 135 microsec
64000 elements => 156 microsec
128000 elements => 194 microsec
256000 elements => 218 microsec
512000 elements => 288 microsec
1024000 elements => 380 microsec
2048000 elements => 562 microsec
4096000 elements => 1073 microsec
8192000 elements => 1938 microsec
16384000 elements => 3618 microsec
32768000 elements => 6703 microsec
65536000 elements => 13188 microsec
131072000 elements => 25243 microsec
```

```
Enter number of threads:
4
 1000 elements => 248 microsec
 2000 elements => 277 microsec
 4000 elements => 271 microsec
 8000 elements => 305 microsec
16000 elements => 386 microsec
32000 elements => 378 microsec
64000 elements => 405 microsec
128000 elements => 523 microsec
256000 elements => 499 microsec
512000 elements => 666 microsec
1024000 elements => 907 microsec
2048000 elements => 855 microsec
4096000 elements => 1568 microsec
8192000 elements => 2224 microsec
16384000 elements => 3929 microsec
32768000 elements => 6810 microsec
65536000 elements => 13254 microsec
131072000 elements => 24836 microsec
```

```
Enter number of threads:
8
 1000 elements => 451 microsec
 2000 elements => 561 microsec
 4000 elements => 576 microsec
 8000 elements => 722 microsec
16000 elements => 807 microsec
32000 elements => 1159 microsec
64000 elements => 1829 microsec
128000 elements => 2064 microsec
256000 elements => 2096 microsec
512000 elements => 2481 microsec
1024000 elements => 3196 microsec
2048000 elements => 3889 microsec
4096000 elements => 3073 microsec
8192000 elements => 5509 microsec
16384000 elements => 9776 microsec
32768000 elements => 13158 microsec
65536000 elements => 21601 microsec
131072000 elements => 31766 microsec
```

How many threads?

you can find out how many CPUs/cores your machine has

```
int cores = Runtime.getRuntime().availableProcessors();
```

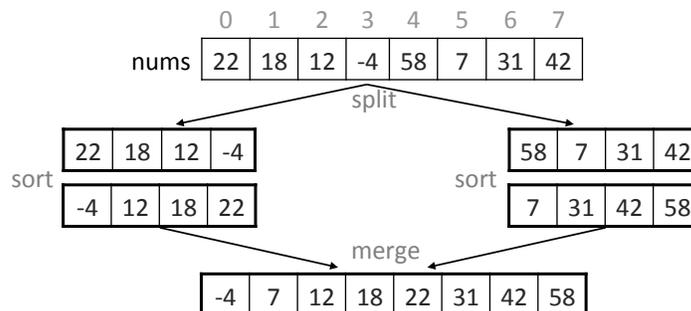
- timings on previous slide were on a computer with 8 cores
- note that a user program may not have access to all cores
 - in previous timings, 8 threads actually degraded performance
- there is overhead associated with creating and managing threads
 - 1 thread was faster than 2 threads up to 1M numbers
 - 2 threads was faster than 4 threads up to 1B numbers

21

Example: parallel mergeSort

suppose we want to take advantage of multicores when sorting

- have separate threads sort the left and right halves, then merge



- each of the threads can spawn new threads, *up to some limit*

22

Example: parallel mergeSort

```
public class SortThread extends Thread {
    private int[] nums;
    private int minIndex;
    private int maxIndex;
    private int threadCount;

    public SortThread(int[] nums, int minIndex, int maxIndex, int threadCount) {
        this.nums = nums;
        this.minIndex = minIndex;
        this.maxIndex = maxIndex;
        this.threadCount = threadCount;
    }

    public void run() {
        MergeSort.mergeSortConcurrently(this.nums, this.minIndex,
                                       this.maxIndex, this.threadCount);
    }
}
```

23

Example: parallel mergeSort

```
public class MergeSort {
    public static void mergeSortConcurrently(int[] a, int threadCount) {
        MergeSort.mergeSortConcurrently(a, 0, a.length-1, threadCount);
    }

    public static void mergeSortConcurrently(int[] a, int minIndex, int maxIndex,
                                           int threadCount) {
        if (minIndex < maxIndex) {
            int mid = (minIndex+maxIndex)/2;
            if (threadCount > 1) {
                Thread leftThread = new SortThread(a, minIndex, mid, threadCount/2);
                Thread rightThread = new SortThread(a, mid+1, maxIndex, threadCount/2);
                leftThread.start();
                rightThread.start();

                try {
                    leftThread.join();
                    rightThread.join();
                } catch (InterruptedException ie) {}
            }
            else {
                MergeSort.mergeSortConcurrently(a, minIndex, mid, threadCount/2);
                MergeSort.mergeSortConcurrently(a, mid+1, maxIndex, threadCount/2);
            }
            MergeSort.merge(a, minIndex, maxIndex);
        }
    }
}
```

24

Example: parallel mergeSort

```

public static void main(String[] args) {
    Random randy = new Random();
    int size = 1000;

    System.out.println("Enter the thread limit: ");
    Scanner input = new Scanner(System.in);
    int numThreads = input.nextInt();
    input.close();

    while (true) {
        int[] nums = new int[size];
        for (int j = 0; j < size; j++) {
            nums[j] = randy.nextInt();
        }

        long start = System.currentTimeMillis();
        MergeSort.mergeSortConcurrently(nums, numThreads);
        long end = System.currentTimeMillis();

        System.out.printf("%10d elements => %6d ms \n", size, end-start);
        size *= 2;
    }
}

```

25

```

Enter the thread limit:
1
1000 elements => 2 ms
2000 elements => 2 ms
4000 elements => 1 ms
8000 elements => 1 ms
16000 elements => 2 ms
32000 elements => 5 ms
64000 elements => 10 ms
128000 elements => 20 ms
256000 elements => 36 ms
512000 elements => 76 ms
1024000 elements => 162 ms
2048000 elements => 340 ms
4096000 elements => 695 ms
8192000 elements => 1428 ms
16384000 elements => 2895 ms
32768000 elements => 5988 ms
65536000 elements => 12479 ms
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

```

```

Enter the thread limit:
2
1000 elements => 1 ms
2000 elements => 1 ms
4000 elements => 1 ms
8000 elements => 3 ms
16000 elements => 3 ms
32000 elements => 7 ms
64000 elements => 14 ms
128000 elements => 17 ms
256000 elements => 22 ms
512000 elements => 47 ms
1024000 elements => 103 ms
2048000 elements => 197 ms
4096000 elements => 392 ms
8192000 elements => 790 ms
16384000 elements => 1593 ms
32768000 elements => 3327 ms
65536000 elements => 6681 ms

```

```

Enter the thread limit:
4
1000 elements => 2 ms
2000 elements => 2 ms
4000 elements => 2 ms
8000 elements => 2 ms
16000 elements => 4 ms
32000 elements => 6 ms
64000 elements => 14 ms
128000 elements => 21 ms
256000 elements => 14 ms
512000 elements => 43 ms
1024000 elements => 73 ms
2048000 elements => 136 ms
4096000 elements => 252 ms
8192000 elements => 515 ms
16384000 elements => 1055 ms
32768000 elements => 2232 ms
65536000 elements => 4193 ms

```

```

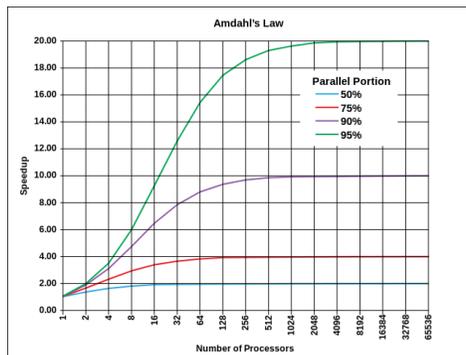
Enter the thread limit:
8
1000 elements => 3 ms
2000 elements => 3 ms
4000 elements => 2 ms
8000 elements => 4 ms
16000 elements => 3 ms
32000 elements => 5 ms
64000 elements => 9 ms
128000 elements => 18 ms
256000 elements => 30 ms
512000 elements => 29 ms
1024000 elements => 62 ms
2048000 elements => 114 ms
4096000 elements => 205 ms
8192000 elements => 401 ms
16384000 elements => 804 ms
32768000 elements => 1725 ms
65536000 elements => 3303 ms

```

Amdahl's Law

the speedup that can be achieved by parallelizing a program is limited by the sequential fraction of the program.

- e.g., if 25% of the program must be performed sequentially, you can only get a 4x speedup (no matter how many processors)
- part of the trick becomes learning how to minimize the portion of the program that must be performed sequentially



27