CS-206 Concurrency

Lecture 13

Wrap Up

Spring 2015

Prof. Babak Falsafi

parsa.epfl.ch/courses/cs206/

Created by Nooshin Mirzadeh, Georgios Psaropoulos and Babak Falsafi

Where are We?

Lecture & Lab

M	T	W	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb
2-Mar	3-Mar	4-Mar	5-Mar	6-Mar
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr	28-Apr	29-Apr	30-Apr	1-May
4-May	5-May	6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	19-May	20-May	21-May	22-May
25-May	2[illegible]	27-May	28-May	29-May

- Wrap up
  - Concurrent list
  - Concurrent hash tables
  - Scheduling
  - Work distribution
  - Data parallel computing
  - GPU

Lecture 8: Concurrent Lists

Synchronization Patterns:

- With locks
  - Coarse-grained
  - Fine-grained
  - Optimistic
  - Lazy
- Lock-free

You should be able to:
- Describe each pattern
- Argue about pros & cons
- Prove concurrency properties

Exercise 1

- In the lock-free algorithm, argue for and against having the `contains()` method help in the cleanup of logically removed entries.

Exercise 1: Answer

- Assumption: most method calls are to `contains()`
- Pros
  - Faster `add()`/`delete()`
- Cons
  - Synchronization added → latency penalty for `contains()`
  - Slowdown depending on the method call breakdown

Lecture 9: Concurrent Hash Tables

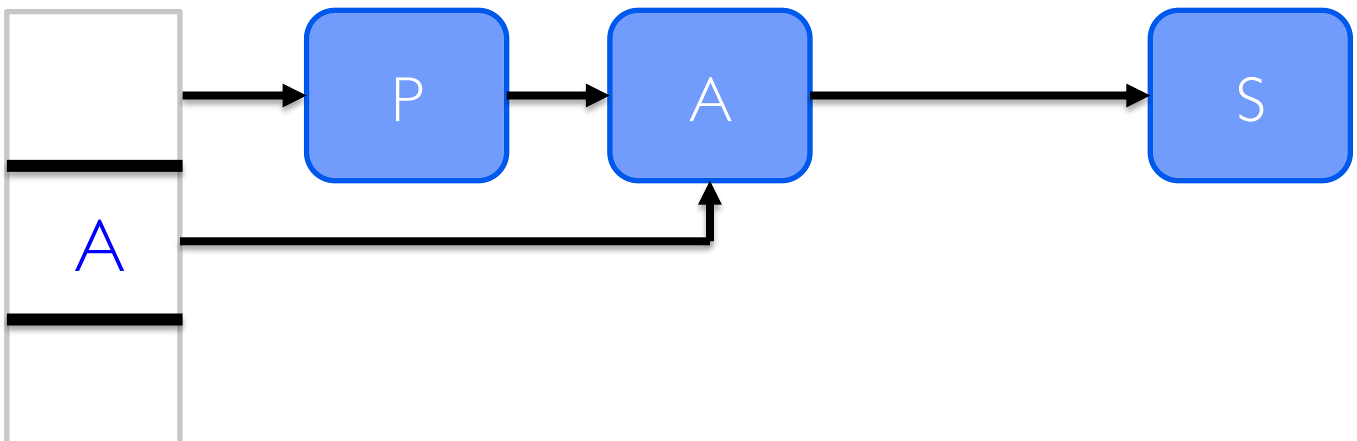
- Coarse-Grained Locks
- Fine-Grained Locks
- Striped Locks
- Lock-free
  - Sentinels

You should be able to:
- Describe each implementation
- Argue about pros & cons

Exercise 2

- For the lock-free HashSet, show an example of the problem that arises when deleting an entry pointed to by a bucket reference, if we do not add a *sentinel* entry to the start of each bucket.

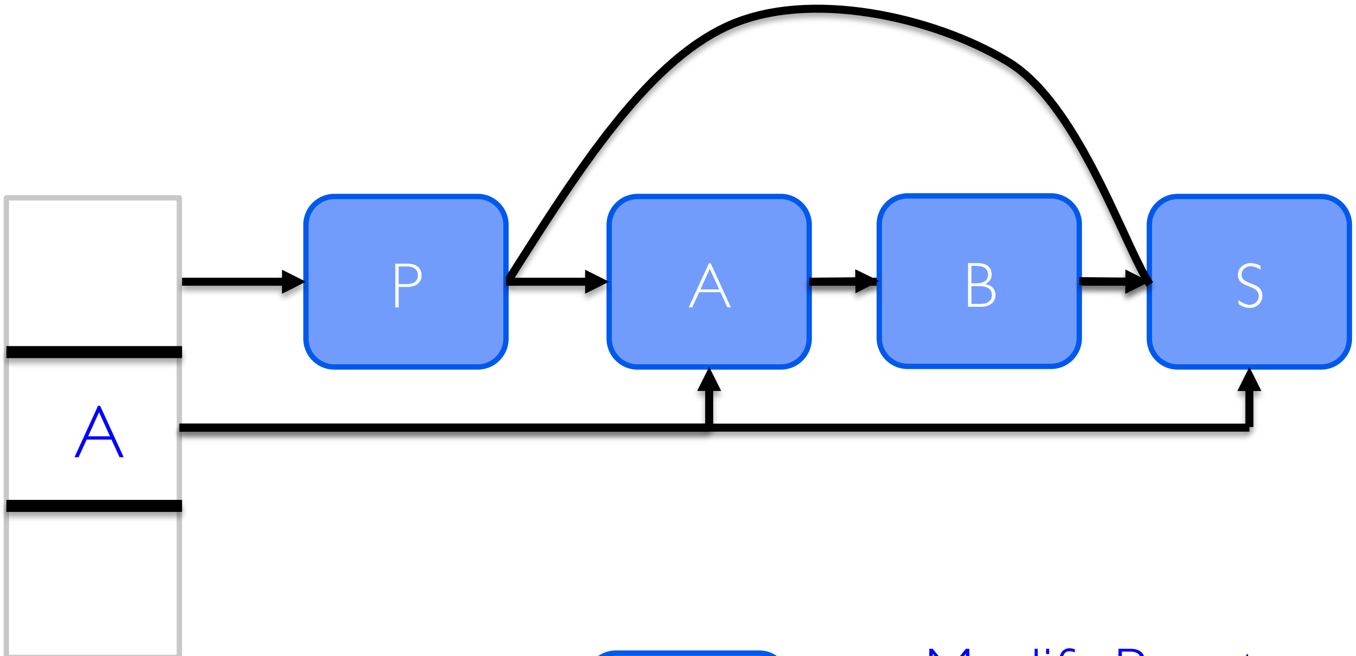
Exercise 2: Answer



Concurrently:
- Remove A
- Insert B after A

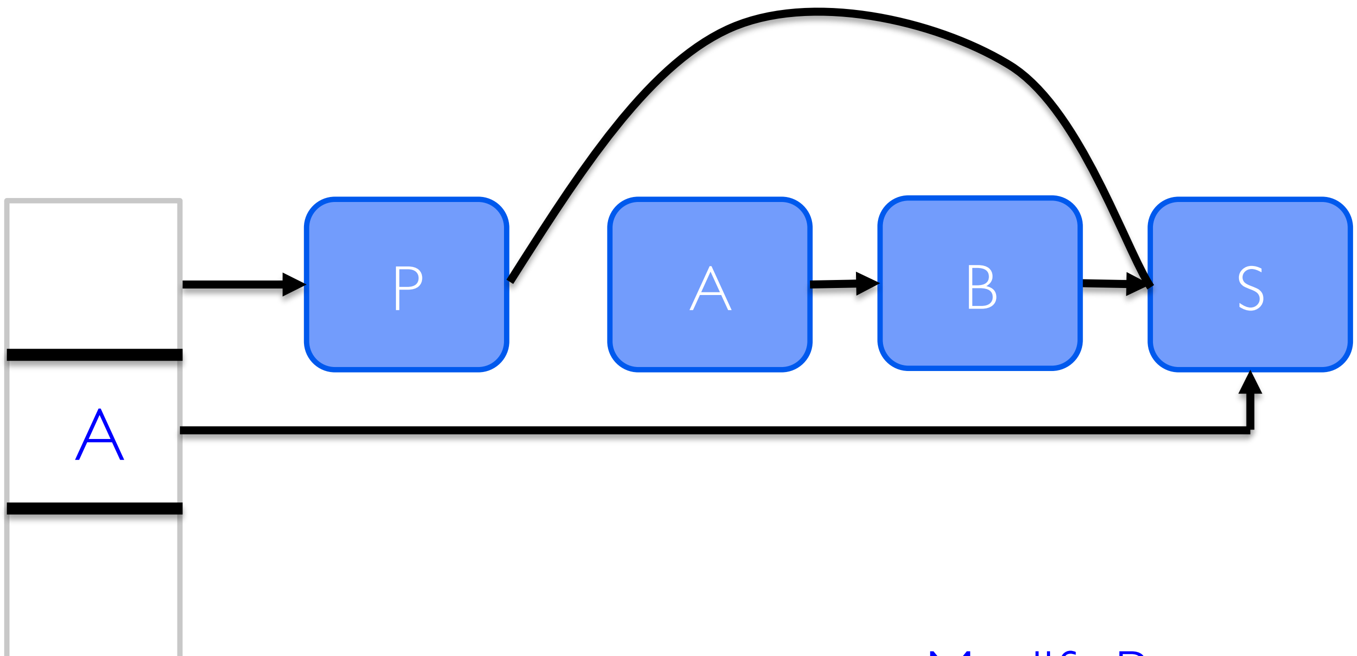
P.next and bucket A.head are not changed atomically together

Exercise 2: Answer



a. Modify P.next
b. Modify A.next
c. Modify bucket A.head

Exercise 2: Answer



a. Modify P.next
b. Modify A.next
c. Modify bucket A.head

Lecture 10: Scheduling

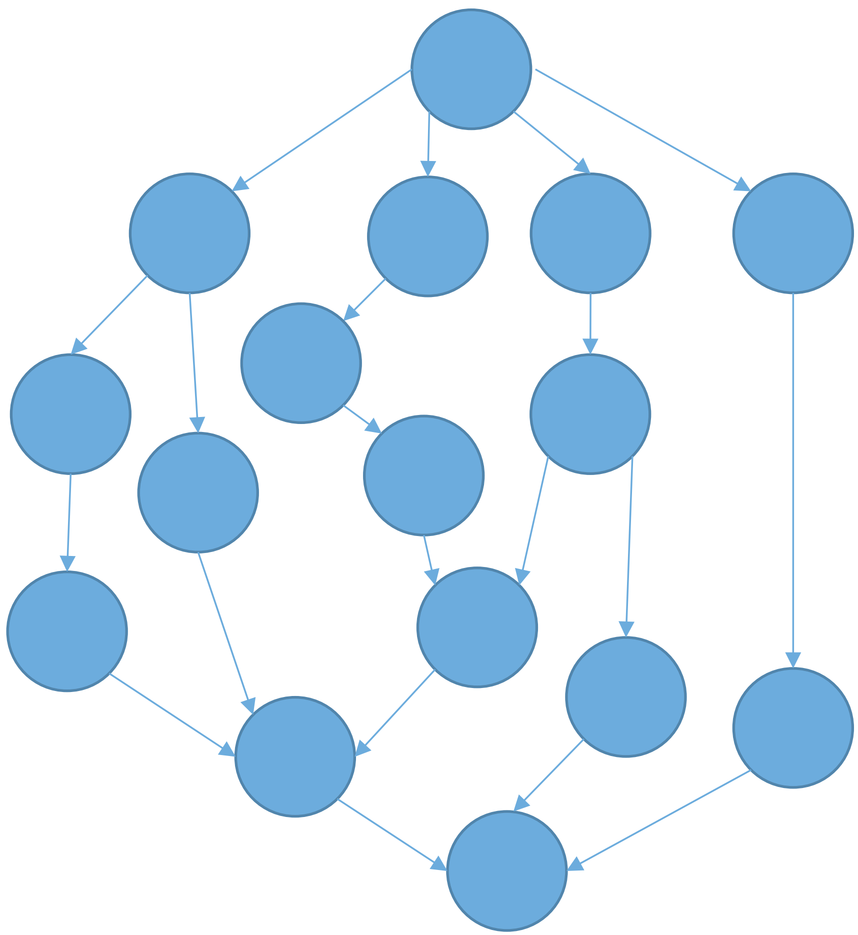
- Thread pools vs threads
  - ExecutorService
  - Runnable/Callable
  - Future
- DAG Model
  - T_1, T_p, T_∞
  - Work & Critical Path
  - Speedup

You should be able to:
- Write code using them

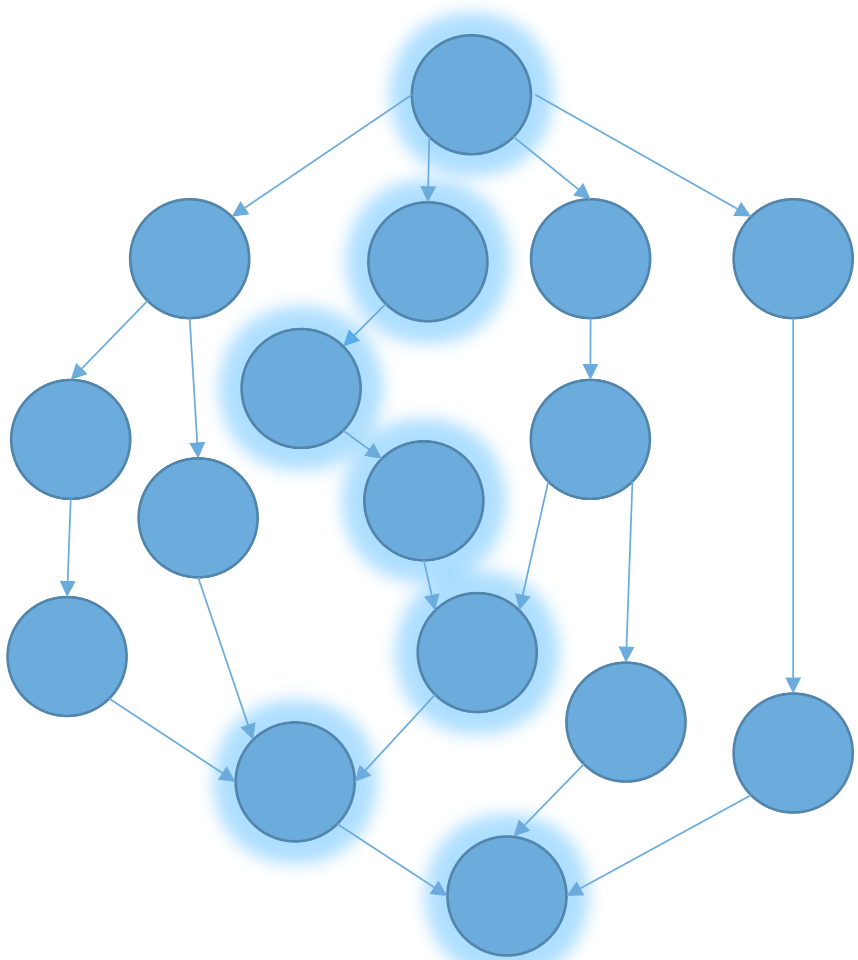
You should be able to:
- Draw the DAG
- Calculate Ts & speedup
- Explain laws & metrics

Exercise 3

- Calculate T_1, T_∞ and the max speed up for the DAG.



Exercise 3: Answer



$T_1 = 16$
$T_\infty = 7$
Speedup = 16/7

Lecture 10: Work Distribution

- Lock-free work stealing
  - Each thread has a pool of ready work (Each work pool is a double ended queue)
  - Remove work without synchronizing
  - If you run out of work, steal someone else's
  - Choose victim at random
- Work balancing
  - Each thread periodically *balance* its workloads with a randomly chosen partner
  - Lightly-loaded threads more likely to initiate rebalancing

Exercise 4

- In the `popBottom()` method of class `BDEQueue`, the `bottom` field is volatile to assure that in `popBottom()` the decrement `bottom--` is immediately visible. Describe a scenario that explains what could go wrong if `bottom` were not declared as volatile.

Exercise 4: popTop()

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp)) return r;
 return null;
 }
```

Exercise 4: popBottom()

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
 bottom = 0; }
```

Exercise 4: Answer

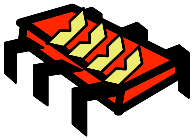
- bottom == k+1, top == k
- Thread 1: bottom-- → bottom = k
- Thread 2: bottom == k+1, oldTop == k

→ Thread 2 won CAS → steal tasks[k], top == k+1

- Thread 1: r = tasks[k], oldTop = k+1

→ Return tasks[k] ✗

Lecture 11: Data Parallel Computing

- Vector Processors: SIMD 
  - High-level operations work on linear arrays of numbers: "vectors"
  - Vector reduces ops by 1.2X, instructions by 20X
- Graphics Processing Units (GPUs): SIMT
  - Thousands of tiny cores, mostly ALU, little cache
  - Integrated vs. discrete
  - Lightweight threads
  - Programming language: CUDA

Lecture 11: GPU (1/2)

- Programmer's view
  - CPU: host, GPU: device
  - Create data in CPU and copy to GPU mem
  - Launch GPU kernel
  - Synchronize CPU and GPU, copy results back to CPU

- Per Kernel Computation Partitioning
  - Grid, blocks, and threads
  - Threads within a block can communicate/synchronize
  - Threads across blocks can't communicate

Lecture 11: GPU (2/2)

- Memory model
  - Global memory: Communicating R/W data between host and device
  - Texture and Constant Memories: Constants initialized by host

- Execution Model: Ordering
  - Execution order is undefined

Exercise 5

- You are writing a CUDA kernel to do vector addition. However, instead of mapping one vector element to each thread, you are mapping two vector elements to each thread. Show the code for this kernel.
- You are writing the C host code to invoke the kernel you wrote in previous part for a vector of 5,000 elements. For your version of CUDA, the maximum block size is 1024. How many blocks will you create, and how many threads per block will you use?

Exercise 5: Answer

```
__global__ void vadd(int *a, int *b, int *c, int N){
      int i = blockIdx.x * 2*blockDim.x + threadIdx.x;
      int j = i + blockDim.x;
      c[i] = a[i] + b[i];
      if (j < N) c[j] = a[j] + b[j];
}
```

Exercise 5: Answer

```
int main(){
  int N = 5000;  int a[N], b[N], c[N];
  int *d_a, *d_b, *d_c; int SIZE = N*sizeof(int);
  cudaMalloc ((void **) &d_a, SIZE); …
  cudaMemcpy (d_a, a, SIZE, cudaMemcpyHostToDevice);
  …
  dim3 gridDim(ceil(N/(2*128.0)),1,1);
  dim3 blockDim(128,1,1);
  vadd<<< gridDim, blockDim >>> (d_a, d_b, d_c, N);
  cudaDeviceSynchronize ();
  cudaMemcpy (c, d_c, SIZE, cudaMemcpyDeviceToHost));
  CUDA_SAFE_CALL (cudaFree (d_a));
  …}
```

Lecture 12: CUDA

- Matrix multiplication
  - Simple: each thread calculates one element of the result matrix
  - Tiled: Use shared memory to reuse data
- Warp: threads are grouped to run together
  - Warp grouping follows sequential thread id (32 threads)

Lecture 12: CUDA: Reduction (1/2)

- #1: Each thread loads one element into shared memory
  - Reduce: proceed in logN steps
  - Divergent in warp threads
- #2: Replace the divergent branching code with strided index and non-divergent branch
  - 2-way bank conflict
- #3: Replace stride indexing in the inner loop With reversed loop and threadID-based indexing
  - Bad resource utilization

Lecture 12: CUDA: Reduction (2/2)

- #4: Read and reduce the first two elements
  - Memory bandwidth is still underutilized

- #5: Unrolling the last warp

Exercise 6

- For the kernel that you wrote in Exercise 5, do you expect any of the warps (groups of 32 threads with consecutive IDs) to take divergent paths through the code? If so, how many, and how do you expect this divergence to affect the performance of the kernel?

Exercise 6: Answer

- 5000 % 256 = 136 → 136 elements in the last block
- 136 – 128 = 8 → First warp takes divergent path