COMP 322: Fundamentals of Parallel Programming

Lecture 19: Task Scheduling Policies

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

http://comp322.rice.edu/

Worksheet #17: Cooperative vs Blocking Runtime scheduler

Assume that creating an async causes the task to be pushed into the work queue for execution by any available idle thread.

Fill the following table for the program shown on the right by adding the appropriate number of threads required to execute the program. For the minimum or maximum numbers, your answer must represent a schedule where at some point during the execution all threads are busy executing a task or blocked on some synchronization constraint.

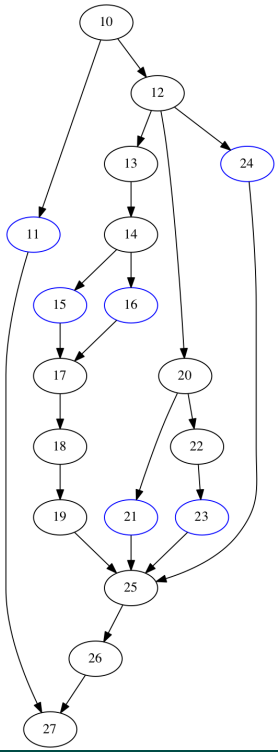
	Minimum number of threads	Maximum number of threads
Cooperative Runtime	1	?
Blocking Runtime	?	?

```
10. finish {
11.   async { S1; }
12.   finish {
13.     async {
14.       finish {
15.         async { S2; }
16.         S3;
17.       }
18.       S4;
19.     }
20.     async {
21.       async { S5; }
22.       S6;
23.     }
24.     S7;
25.   }
26.   S8;
27. }
```


Worksheet #17: Cooperative vs Blocking Runtime scheduler

```
10. finish {
11.   async { S1; }
12.   finish {
13.     async {
14.       finish {
15.         async { S2; }
16.         S3;
17.       }
18.       S4;
19.     }
20.     async {
21.       async { S5; }
22.       S6;
23.     }
24.     S7;
25.   }
26.   S8;
27. }
```

Maximum threads: If we proceed through the graph in top-down manner incrementally, how many maximum leaf nodes can we have?

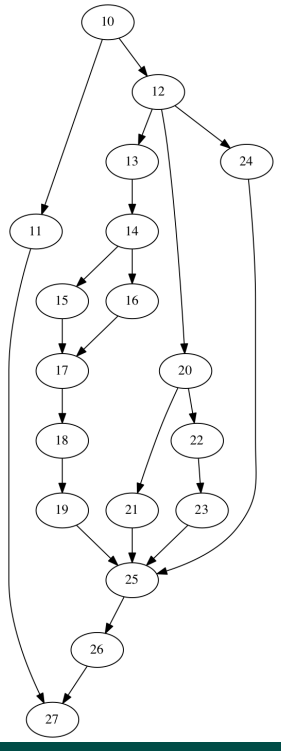


	Maximum number of threads
Cooperative Runtime	6
Blocking Runtime	6

	Minimum number of threads
Cooperative Runtime	1
Blocking Runtime	?

Worksheet #17: Cooperative vs Blocking Runtime scheduler

```
10. finish {
11.   async { S1; }
12.   finish {
13.     async {
14.       finish {
15.         async { S2; }
16.         S3;
17.       }
18.       S4;
19.     }
20.     async {
21.       async { S5; }
22.       S6;
23.     }
24.     S7;
25.   }
26.   S8;
27. }
```

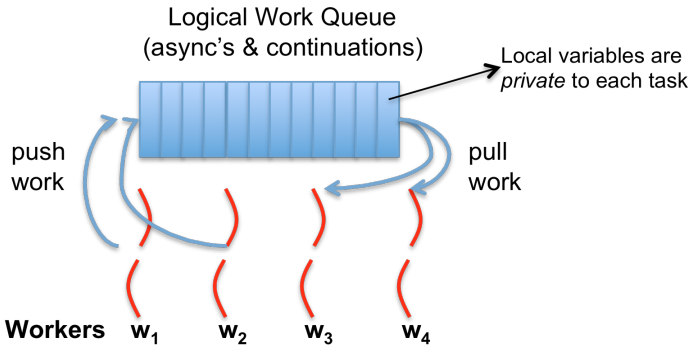


work pool
11
13|20
21
21|15
21|15
15

Start time	T1	T2	T3
0	10		
1	12	11	
2	24	13	20
3	25	14	22
4	25	16	23
5	25	17	21
6	25	17	15
7	25	18	
8	25	19	
9	25		
10	26		
11	27		
13			

	Minimum number of threads
Blocking Runtime	3

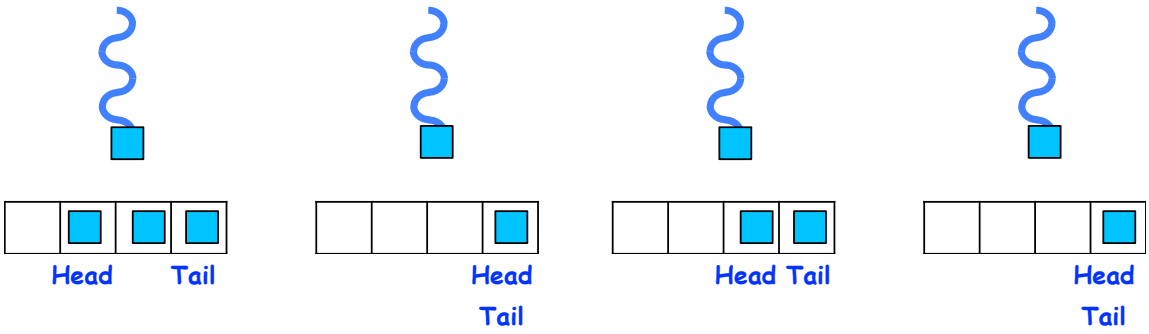
Work-Sharing Scheduling Paradigm



- Busy worker eagerly distributes (*shares*) new work
- Idle worker retrieves work from the task pool
- Easy implementation with global task pool
- Access to the global pool needs to be synchronized: *scalability bottleneck*

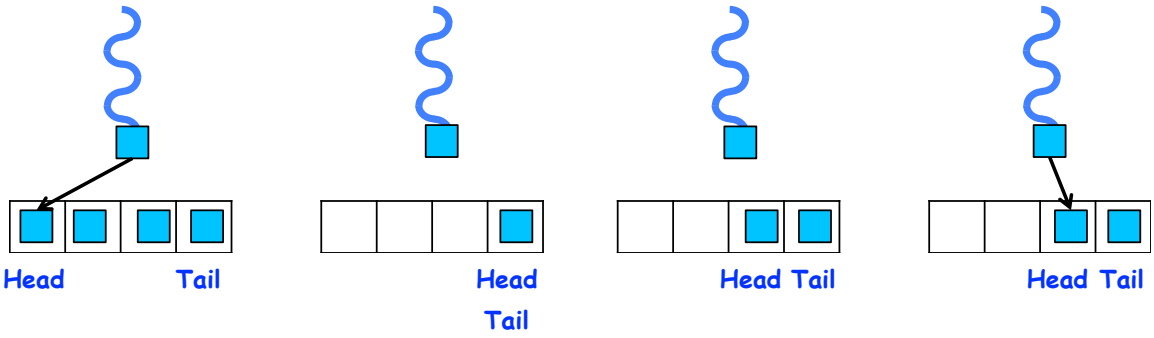
Work Stealing Scheduling Paradigm

- The threads in a ForkJoinPool attempt to *dynamically balance the load* of work among them via "work stealing"
- Each worker thread keeps a double-ended queue (*dequeue*)
  - insertion or removals of elements at the front (head) or the end (tail)



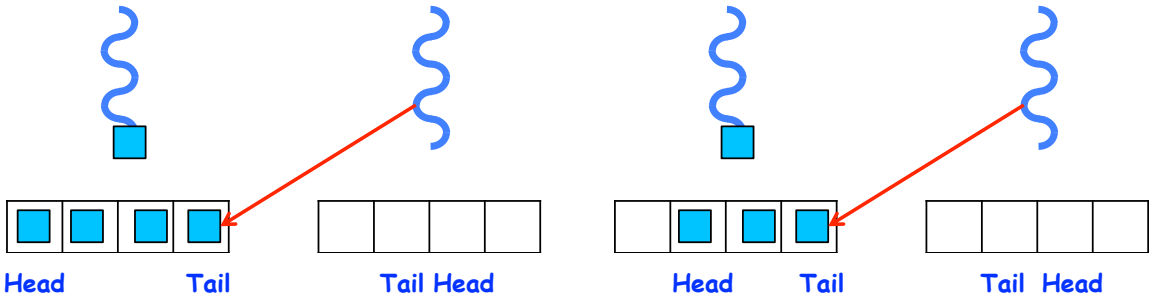
A Worker Performs a Step of Work By:

- Taking a `task` T off the front of its dequeue
- Calling the `compute` method of T
- Inserting any recursive subtasks of T on the front of its dequeue



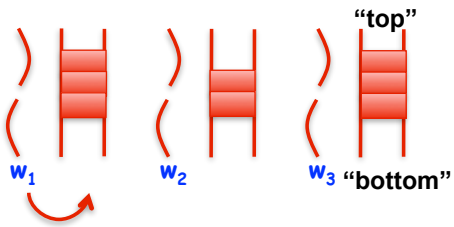
If a Worker Thread's Dequeue is Empty

- It removes a `task` off the end of *another* worker thread's dequeue
- Requires the dequeues to be thread-safe (why?)
- Removing from the end assures the largest tasks are taken (why?)
  - Minimizes interaction among threads



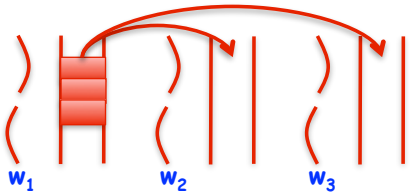
Work-first vs. Help-first work-stealing policies

- When encountering an async
  - Help-first policy
    - Push async on "bottom" of local queue, and execute next statement
  - Work-first policy
    - Push continuation (remainder of task starting with next statement) on "bottom" of local queue, and execute async
- When encountering the end of a finish scope
  - Help-first policy & Work-first policy
    - Store continuation for end-finish
      - Will be resumed by last async to complete in finish scope
    - Pop most recent item from "bottom" of local queue
    - If local queue is empty, steal from "top" of another worker's queue
- Current HJ-lib runtime only supports help-first policy



Local push/pop by w_1

Stealing by w_2 and w_3



Work-first vs. Help-first work-stealing policies on 2 processors (contd)

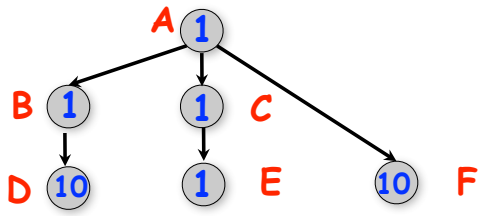
```
1. finish {
2.   // Start of Task T0 (main program)
3.   sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
4.   async { // Task T1 computes sum of upper half of array
5.   for(int i=X.length/2; i < X.length; i++)
6.   sum2 += X[i];
7.   }
8.   // T0 computes sum of lower half of array
9.   for(int i=0; i < X.length/2; i++) sum1 += X[i];
10.  }
11. // Task T0 waits for Task T1 (join)
12. return sum1 + sum2;
13.} // finish
```

Help-First worker does not switch tasks
Work-First worker will switch tasks

Continuations

Help-First worker can switch tasks
Work-First worker can switch tasks

Scheduling Program Q1 using a Help-First Work-Stealing Scheduler



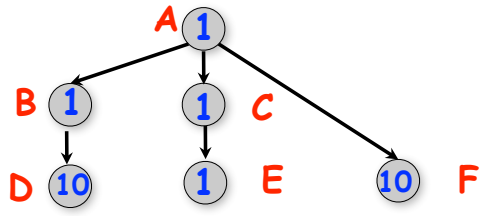

```
1. // Program Q1
2. A; // Executes on P1
3. finish {
4. // P1 pushes 6, which is then
5. // stolen by P2
6. async { B; D; }
7. // P1 pushes 8
8. async F;
9. // P1 pushes 10
10.  async { C; E; }
11. }
12. // P1 stores continuation and pops 10
13. // P1 pops 8
```

deque-1
6|8|10
8
8
-
-
-
-
-
-
-
-
-
-
-
-
-

Thrd1	Start time	Thrd2
A	0	
C	1	B
E	2	D
F	3	D
F	4	D
F	5	D
F	6	D
F	7	D
F	8	D
F	9	D
F	10	D
F	11	D
F	12	
	13	

deque-2
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

Scheduling Program Q1 using a Work-First Work-Stealing Scheduler




```
1. // Program Q1
2. A; // Executes on P1
3. finish {
4. // P1 pushes continuation for 9,
5. // and executes 6
6. async { B; D; }
7. // P2 pushes continuation for 11,
8. // and executes 9
9. async F;
10. // P2 executes 11
11.  async { C; E; }
12. }
```

deque 1
9
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

Thrd1	Start time	Thrd2
A	0	
B	1	F
D	2	F
D	3	F
D	4	F
D	5	F
D	6	F
D	7	F
D	8	F
D	9	F
D	10	F
D	11	C
	12	E
	13	

deque 2
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-