Concurrency: Multi-core Programming & Data Processing

Lab 11

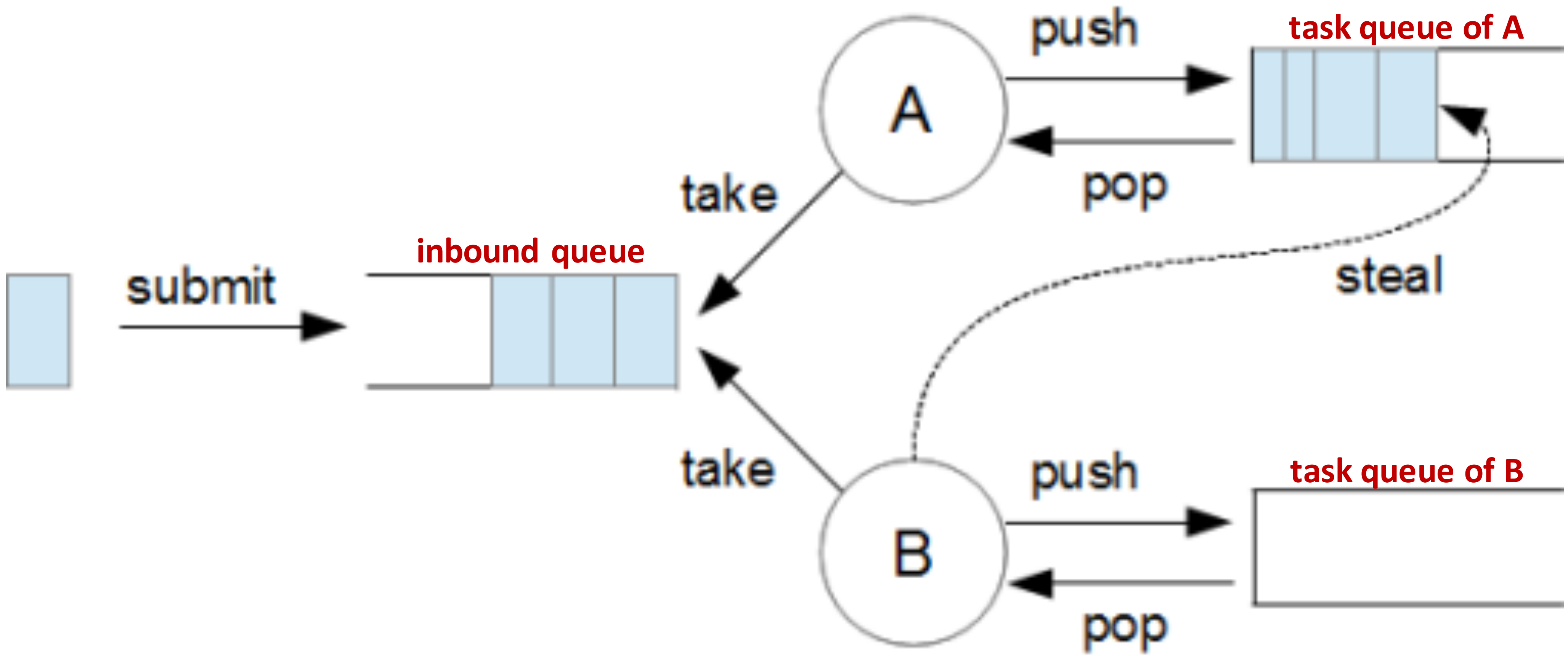
-- ForkJoin --

ForkJoin framework

- **ForkJoinPool:** ThreadPoolExecutor++
  - Less competition between threads
  - More collaboration between threads
  - One instance to rule (run) them (the tasks) all!
- Types of tasks:
  - `RecursiveTask<V>`: returns a result
  - `RecursiveAction`: doesn't return anything
  - `ForkJoinTask<V>`: superclass of the other two, <u>never</u> use it directly

How it works

Double ended. So that stealing can happen at other end



Use cases

- Suitable only for executions where workers schedule new tasks in their own queues
- **Scenario 1:** (recursively) split huge task in small pieces
  - room for work stealing
  - compute directly small sub-tasks (size has to amortize the overhead of the framework!)
- **Scenario 2:** tasks schedule follow-up tasks
  - tasks are self-contained
  - models event-driven processes

Other considerations

Rule: call `fork()` early and `join()` late, do useful work in-between

- `fork()` starts async parallel computation, returns immediately
- `join()` blocks until the computation ends, returns result
- If `join()` called immediately: no parallelism

Other considerations

Rule: create more tasks than processors

- Framework's job to schedule efficiently your tasks
- Number of available processors can vary
- Takes advantage of work-stealing&Co.

Other considerations

Rule: tasks shouldn't be too small or too big

- If too little to process => more overhead in running the framework than processing
- If too much to process => close to the sequential version, or at least to a ThreadPoolExecutor with extra overhead

Other considerations

Rule: two recursive sub-tasks – call `fork()` only once

- Better call `fork()` for one sub-task and `compute()` for the other (see example in PDF)
- Have `fork()` call recursively `fork()` and `compute()` and so on
- Otherwise, <u>a lot less efficient</u>

Other considerations

Rule: order of instructions does matter

```
left.fork();
long rightAns = right.compute();
long leftAns  = left.join();
return leftAns + rightAns;
```

- versus

```
left.fork();
long leftAns  = left.join();
long rightAns = right.compute();
return leftAns + rightAns;
```

- or

```
long rightAns = right.compute();
left.fork();
long leftAns  = left.join();
return leftAns + rightAns;
```

Other considerations

Rule: call `invoke()` only in sequential code

- Not in `RecursiveTask` or `RecursiveAction` – still part of the same parallel task, call directly `compute()` and `fork()` there
- `invoke()` is there to start parallelism

Other considerations

Rule: catch exceptions inside `compute()`

- Exceptions in fork-join can be confusing
- `compute()` happens in a different thread than the caller (that actually called `fork()`) => convoluted stack trace with extra library calls

Exercise

- Write a Java program that solves the "word count" problem using the ForkJoin framework. The program gets a file name as a parameter and returns the first 10 most encountered words in the text. The program uses all available processors (e.g., returned by `Runtime.getRuntime().availableProcessors()`).