Relatório do Projeto de Programação Concorrente

João Varelas

June 22, 2020

Contents

Filas concorrentes

Fila baseada em monitores/*locks*

Capacidade fixa (`MBQueue.java`)

Um dos blocos `synchronized` foi eliminado de forma que `size++` fique incluído num só bloco.

Alteração de `notify()` para `notifyAll()` para que todas as threads sejam notificadas. No caso de `notify()` apenas uma thread é notificada e esta escolha é feita de forma não-determinista (exceções `CWaitDeadlockError`).

Capacidade ilimitada (`MBQueueU.java`)

Estende a classe anterior. O método `add()` verifica se o array está cheio e cria uma nova cópia com o dobro do tamanho (*unbounded*).

Fila baseda em STM

Capacidade fixa (`STMBQueue.java`)

Em `add()`, a instrução `STM.increment` foi colocada dentro do bloco `STM.atomic` imediatamente a seguir a `array.update()`.

No caso de `remove()`, passa a existir apenas uma transação que retorna um elemento da fila.

Capacidade ilimitada (`STMBQueueU.java`)

No método `add()`, caso a fila esteja cheia, é criado um novo array com o dobro do tamanho do tipo `TArray.View<E>`. Os elementos do array original passam para a nova cópia através de `newArray.update(idx, elem)`.

O método `remove()` é semelhante ao anterior. A instância do array é obtida através de `arrayRef.get()`.

Fila baseada em primitivas atómicas

Capacidade fixa (`LFBQueue.java`)

Nesta implementação as threads entram em `rooms` de execução para `add()`, `remove()` e `size()`. Várias threads na mesma sala podem efetuar a mesma operação de forma concorrente. Threads que tentam entrar noutras salas irão bloquear até que as salas ocupadas fiquem livres eventualmente (progresso não-bloqueante).

Capacidade ilimitada (`LFBQueueU.java`)

Estende a classe anterior. O método `add()` redimensiona o array caso esteja cheio e utiliza uma flag do tipo `AtomicBoolean` para que outras threads na sala correspondente à execução de `add()` não acedam ao array enquanto estiver a ser redimensionado (exclusão mútua).

Backoff

Mitigação/delay em pontos de espera ativos.

Análise de execução linearizável

Precedência de operações

Cada thread t_1, t_2 e t_3 executa uma sequência de ações.

Sejam as ações:

- $a_1 =$ `q.add(3)`, $a_2 =$ `q.remove()`, $a_3 =$ `a.set(`a_2`)`
- $a_4 =$ `q.remove()`, $a_5 =$ `b.set(`a_4`)`
- $a_6 =$ `q.size()`, $a_7 =$ `q.add(`a_6`)`, $a_8 =$ `q.add(`$a_6 + 1$`)`

As threads que as executam são respetivamente t_1, t_2 e t_3. A precedência entre ações para cada thread é a seguinte:

- t_1: $a_1 \to a_2 \to a_3$
- t_2: $a_4 \to a_5$
- t_3: $a_6 \to a_7 \to a_8$

Possíveis linearizações

As ações a_2 e a_4 irão bloquear a thread caso a fila esteja vazia.

É necessário que seja executada a_1 ou a_7 para que seja possível remover algum elemento existente na fila.

- S_1: $[a_1][a_2][a_3][a_4][a_6][a_7][a_5][a_8]$
- S_2: $[a_1][a_2][a_3][a_4][a_6][a_7][a_8][a_5]$
- S_3: $[a_1][a_2][a_3][a_6][a_4][a_7][a_5][a_8]$
- S_4: $[a_1][a_2][a_3][a_6][a_4][a_7][a_8][a_5]$
- S_5: $[a_1][a_2][a_3][a_6][a_7][a_8][a_4][a_5]$
- $(\dots)$

As possibilidades de valores para os registos `a`, `b`, `c` e `d` são as seguintes:

- $P = \{0, 1, 3\}$
- `a`,`b`,`c` $\in P$
- `d` $\in P \cup \{2\}$

O estado da fila Q_i depende da ordem de execução de ações de cada linearização S_i:

- Q_1: $[\,] \xrightarrow{\text{add(3)}} [3] \xrightarrow{\text{remove()}} [\,] \xrightarrow{\text{add(0)}} [0] \xrightarrow{\text{remove()}} [\,] \xrightarrow{\text{add(1)}} [1]$
- Q_2: $[\,] \xrightarrow{\text{add(3)}} [3] \xrightarrow{\text{remove()}} [\,] \xrightarrow{\text{add(0)}} [0] \xrightarrow{\text{remove()}} [\,] \xrightarrow{\text{add(1)}} [1]$
- $(\dots)$
- Q_5: $[\,] \xrightarrow{\text{add(3)}} [3] \xrightarrow{\text{remove()}} [\,] \xrightarrow{\text{add(0)}} [0] \xrightarrow{\text{add(1)}} [0, 1] \xrightarrow{\text{remove()}} [1]$

Análise de desempenho

- Sistema operativo: Debian 10
- CPU: Intel Core i7 @ 1.80~4.80GHz (4 cores, 8 threads)
- RAM: 8GB

N.º de operações por segundo por thread (em milhares).

Benchmark #1	2 threads	4 threads	8 threads	16 threads	32 threads
Monitor-based	5092.58	839.59	421.90	225.10	105.20
Lock-free backoff=y	8739.40	3578.51	1381.21	529.63	101.44
Lock-free backoff=n	1306.93	273.59	94.60	23.42	8.34
STM	3243.34	164.48	39.02	16.78	7.95

Benchmark #2	2 threads	4 threads	8 threads	16 threads	32 threads
Monitor-based	5383.68	905.36	411.57	229.54	102.37
Lock-free backoff=y	8039.91	2805.01	1136.29	159.34	43.08
Lock-free backoff=n	1581.14	289.04	90.46	31.41	8.87
STM	2144.34	116.95	35.64	15.49	7.33

Benchmark #3	2 threads	4 threads	8 threads	16 threads	32 threads
Monitor-based	4143.75	868.61	377.79	225.80	117.42
Lock-free backoff=y	8272.07	3735.00	1222.58	184.22	276.04
Lock-free backoff=n	1216.58	287.78	85.59	29.54	8.53
STM	1751.17	151.13	38.44	16.53	7.92

Benchmark #4	2 threads	4 threads	8 threads	16 threads	32 threads
Monitor-based	3588.55	848.23	438.81	205.85	104.22
Lock-free backoff=y	5398.35	2869.89	485.43	154.65	43.18
Lock-free backoff=n	1416.27	285.74	106.68	29.64	8.97
STM	691.80	123.26	37.11	15.75	7.21

Benchmark #5	2 threads	4 threads	8 threads	16 threads	32 threads
Monitor-based	5758.43	933.59	417.44	206.06	108.02
Lock-free backoff=y	11407.44	2535.25	1354.40	152.30	44.22
Lock-free backoff=n	1670.72	282.88	93.23	29.74	8.45
STM	1756.67	139.07	41.56	16.42	7.54

Average	2 threads	4 threads	8 threads	16 threads	32 threads
Monitor-based	4793.39	879.07	413.50	218.47	107.45
Lock-free backoff=y	8371.43	3104.73	1115.98	236.02	101.59
Lock-free backoff=n	1438.32	283.80	94.11	28.75	8.63
STM	1917.46	138.97	38.35	16.19	7.59

Web crawler concorrente

São obtidos links da página base (*root*) por via de uma expressão regular. Cada link é colocado sob a forma de sufixo no URL e posteriormente visitado/transferido.

Na implementação concorrente, cada tarefa lançada corresponde a uma transferência. As tarefas são recursivas no sentido de serem lançadas novas tarefas para cada link em sub-páginas.

O `join` das tarefas é realizado a cada nível da árvore de pesquisa (`rid`) para os respetivos `forks` lançados nessa profundidade.

O código de `TransferTask.compute()` é o seguinte:

```
protected Void compute() {
            try {
                List<RecursiveTask<Void>> forks = new LinkedList<>();
                List<String> links = performTransfer(rid, new URL(path));
                URL url = new URL(path);

                for (String link : links) {
                    String newURL = new URL(url, new URL(url, link).getPath()).toString();
                    if (!visited.contains(newURL)) {
                        visited.add(newURL);
                        TransferTask task = new TransferTask(rid + 1, newURL);
                        forks.add(task);
                        task.fork();
                    }
                }

                for (RecursiveTask<Void> task : forks)
                    task.join();

            } catch (Exception e) {
                throw new UnexpectedException(e);
            }
            return null;
        }
```

Análise de desempenho

Tempo de *crawl* decorrido em segundos sobre as páginas de documentação `jdk-8u251-docs-all` (~120MB); média de 5 *benchmarks* sucessivos.

Cada linha da tabela corresponde a uma configuração do *web server* tal que:

- Default threads: 4
- Equal threads: n.º threads do *WS* = n.º threads do *crawler*
- Half threads: n.º threads do *WS* é igual à metade de threads do *crawler*
- Work stealing: flag `WORK_STEALING_POOL` definida `true` no `WebServer.java`

Crawler threads	1 thread	2 threads	4 threads	8 threads	16 threads
WebServer default threads	>10min.	230.8	117.6	48.9	39.3
WebServer equal threads	>10min.	221.4	119.7	45.6	38.0
WebServer half threads	>10min.	226.7	122.6	49.9	39.4
WebServer work stealing on	>10min.	214.8	115.2	44.5	35.2