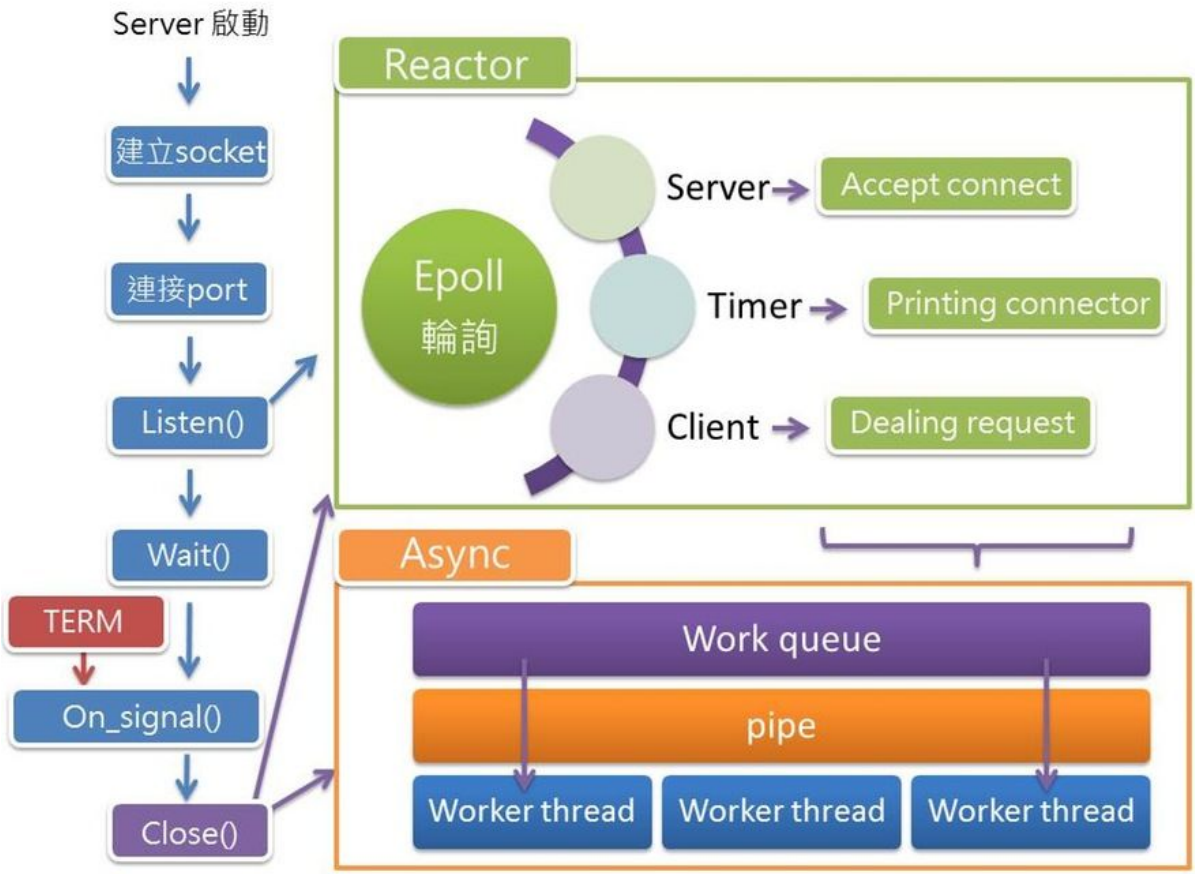
server-framework 研究

contributed by <judy66jo>, <JonSyuGithub>

案例探討: 強化 server-framework 效能



中英文字間請以空白區隔

課程助教

運作流程

1. Listen()執行後將會啟用Reactor處理
    - Accept connect
    - Printing connector
    - Dealing request
2. wait(): thread將會進入thread_join，等待worker thread將工作完成
    - 或者透過TERM (ctrl+c)進入close()

        - close(): 將結束Reactor內的Async裡的所有Worker thread
3. Client的request將由Server的Reactor先送入Work queue並在pipe寫入 one byte
    - work queue為linked list
4. worker thread將會取得pipe的one byte負責工作 (i.e. 處理某個client的request)

async, reactor, protocol-server 彼此之間的關聯

Reactor Pattern

- Resources: client 的 request
- Synchronous Event Demultiplexer: 多對一, Server處理多個Client的要求 i.e. Epoll
- Dispatcher: 將client待處理的request丟入Work queue，並喚醒thread處理這些request
- Request Handler: 也就是Worker thread去處理工作(client request)

async

- 該架構管理thread pool

```
// 處理存入work queue的運行，透過pipe喚醒work thread處理
// 寫入one byte到pipe
static int async_run(async_p async, void (*task)(void *), void *arg)


// 會讀取pipe的one byte
/* listen pipe and release thread */
static void *worker_thread_cycle(void *_async)
```

思考這樣的 API 設計有沒有不足？或者值得做 code refactoring 之處 jserv

reactor

- 回應 requset、使用 epoll 監控整個系統

```
// 將一個工作存入epoll
int set_fd_polling(int queue,
                   int fd,
                   int action,
```

```
                    long milliseconds)


// 透過 reactor_review( ) 將 epoll 中的工作抓出來執行
int reactor_review(struct Reactor *reactor)
```

protocol-server

- 定義 server、連線收發、連線後工作狀態

```
// 用來監控整個系統工作狀態，決定工作是否中斷連線/執行
static void srv_cycle_core(server_pt server)
```

探討 epoll, pipe, signals 等系統呼叫

Epoll

- 非同步I/O，解決因大量連線造成process和thread無法handle的狀況
- 能handle比select更多連線，且非線性掃描，但只能在linux系統上運作
- ET(edge-triggered)：只有在fd變成ready-state時有反應，直到事件結束後又發生下一次，如此一來就不用一直詢問一些不活躍的connect

```
// 生成epoll
int epoll_create(int size)
int epoll_create1(int flags)

// 新增或移除監聽的fd或更改fd的監聽選項
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)

// 回傳ready的fd的數量，把fd存在events array中
int epoll_wait(int epfd, struct epoll_event *events,int maxevents, int timeout)
```

PIPE

- 單向傳輸的資料流，有兩個端口fd，一個是read_only，一個是write_only
- 當嘗試read一個空的pipe時，會被block住，當嘗試寫入full的pipe時，write會被block

```
// 生成pipe，將read,write端fd存入array
int pipe(int pipefd[2])
```

```
// 將count byte的資料寫入pipe_write_end
ssize_t read(int fd, void *buf, size_t count);

// 從pipe_read_end讀count byte的資料
ssize_t write(int fd, const void *buf, size_t count);
```

SIGNAL

- 接收訊號後改變原本動作，確保server正確執行與結束

```
// 改變動作
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);


// The sigaction structure
struct sigaction {
   void     (*sa_handler)(int);
   void     (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t   sa_mask;
   int        sa_flags;
   void     (*sa_restorer)(void);
};
```

- signum 輸入參考

```
Signal     Value     Action   Comment
──────────────────────────────────────────────────────────────────────
──────────────────────────────
SIGINT        2       Term    Interrupt from keyboard
SIGKILL       9       Term    Kill signal
SIGPIPE      13       Term    Broken pipe: write to pipe with no readers
SIGTERM      15       Term    Termination signal
```

apache ab 效能測試

對 Web Server 送出100個 request ，且一次送出32次 request

```
$ ./httpd
// new terminal
$ ab -c 32 -n 100 http://localhost:8080/
```

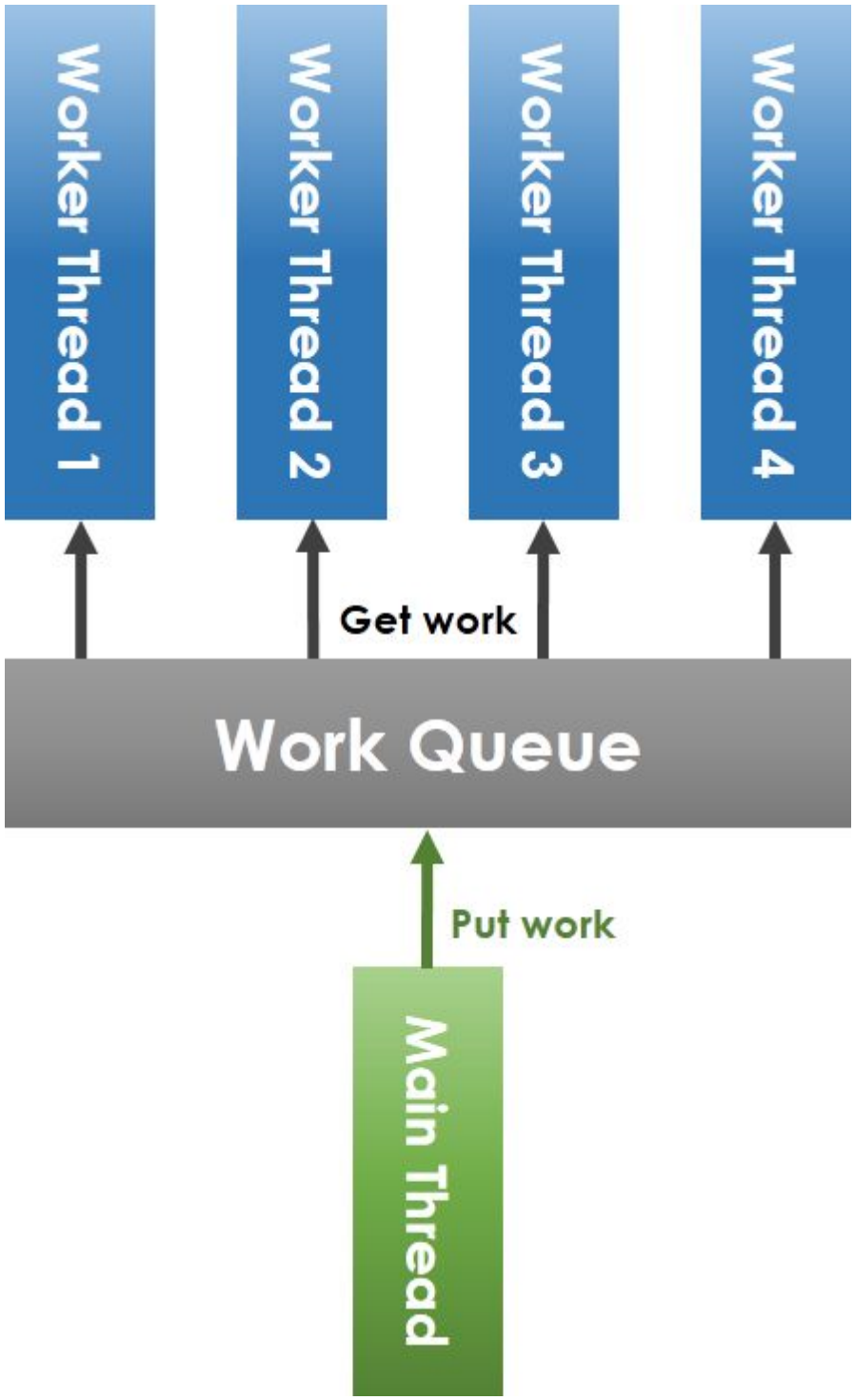
Benchmark

```
This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient).....done


Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /
Document Length:        13 bytes

Concurrency Level:      32
Time taken for tests:   15.691 seconds
Complete requests:      100
Failed requests:        11
   (Connect: 0, Receive: 0, Length: 11, Exceptions: 0)
Total transferred:      9900 bytes
HTML transferred:       1300 bytes
Requests per second:    6.37 [#/sec] (mean)
Time per request:       5021.135 [ms] (mean)
Time per request:       156.910 [ms] (mean, across all concurrent requests)
Transfer rate:          0.62 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.5      1       1
Processing:  3691 3901 144.2   3999    4000
Waiting:        0    0   0.2      0       1
Total:       3691 3901 144.5   4000    4000

Percentage of the requests served within a certain time (ms)
```

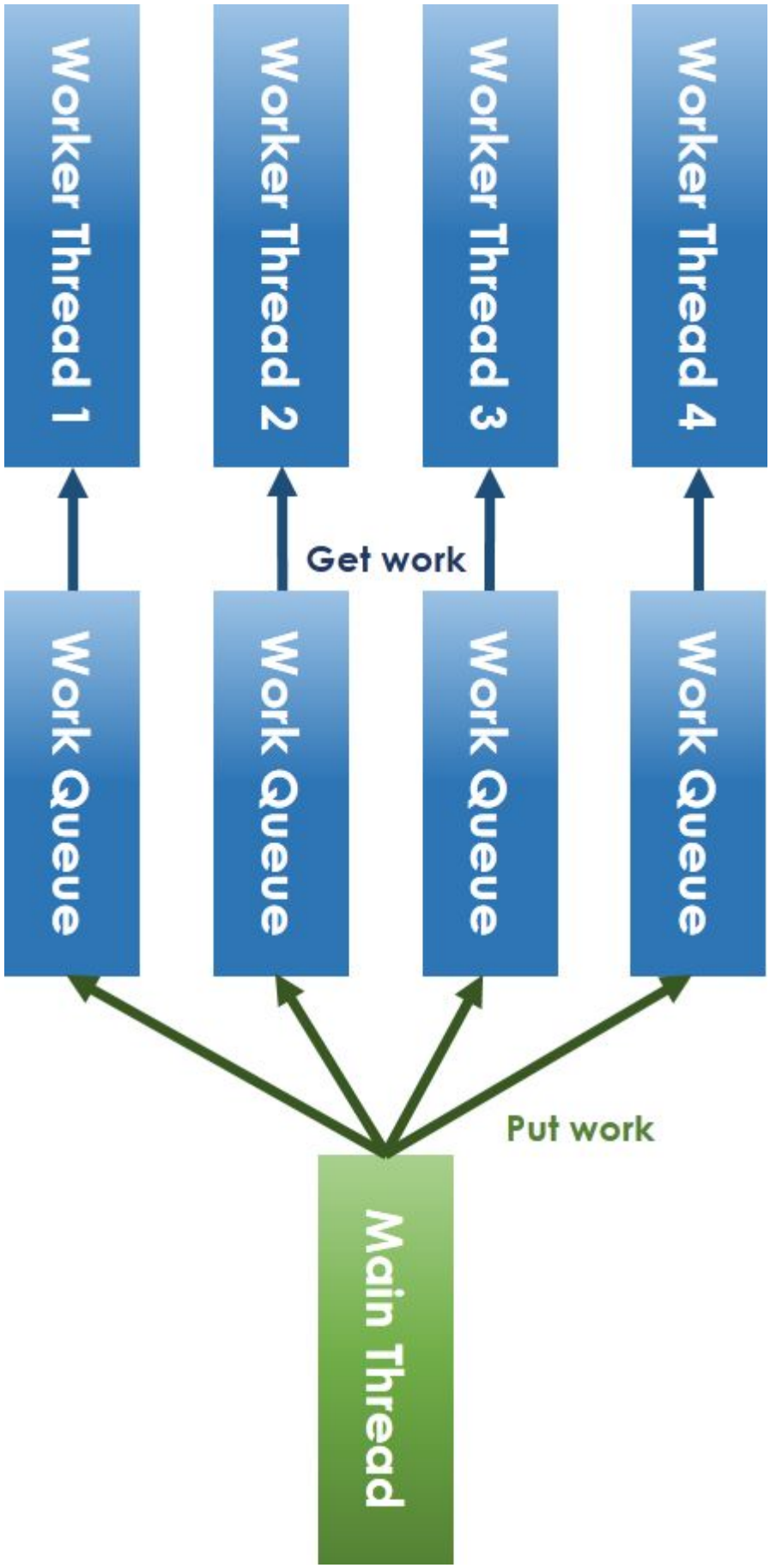
```
 50%   4000
 66%   4000
 75%   4000
 80%   4000
 90%   4000
 95%   4000
 98%   4000
 99%   4000
100%   4000 (longest request)
```

實做與分析 lock-less thread pool

- 在使用多執行緒的程式中，為了避免多個執行緒同時對一個變數進行讀寫的動作(race condition)，會使用 mutex_lock 結合 condition variable。然而在 mutex_lock 的取得及釋出上，有不小的成本。



- 執行緒從一個 work queue 中，去搶（競爭）工作來做，為了避免多個執行緒重複執行同一個工作，在取得工作時都必須以 mutex_lock 將執行緒的執行堵塞住。
- 為了降低 mutex_lock 的使用次數，意味著要降低資源競爭的情況。而為了降低資源競爭的情況，可以將單一個 work queue，改成每一個執行緒都有自己的 work queue，這樣一來每一個執行緒只要從自己的 queue 中拿工作來做，以避免執行緒間搶工作時必須使用 mutex_lock。


Worker Thread 1
Worker Thread 2
Worker Thread 3
Worker Thread 4
Get work
Work Queue
Work Queue
Work Queue
Work Queue
Put work
Main Thread

References

- 案例探討: 強化 server-framework 效能
- 說明 async, reactor, protocol-server 彼此之間的關聯
- 實做與分析 lock-less thread pool