Efficient parallel I/O on multi-core architectures

Adrien Devresse

CERN IT-SDC-ID

Thematic CERN School of Computing 2014

How to make I/O bound application scale with multi-core ?

What is an IO bound application ?

→ A server application

→ A job that accesses big number of files

→ An application that uses intensively network

Stupid example: Simple server monothreaded

```
// create socket
socket_desc = socket(AF_INET , SOCK_STREAM , 0);

// bind the socket
bind(socket_desc,(struct sockaddr *)&server , sizeof(server));
listen(socket_desc , 100);

//accept connection from an incoming client
while(1){
    // declarations
    client_sock = accept(socket_desc, (struct sockaddr *)&client, &c);

    //Receive a message from client
    while( (read_size = recv(client_sock , client_message , 2000 , 0)) > 0{

      // Wonderful, we have a client, do some useful work
      std::string msg("hello bob");
       write(client_sock, msg.c_str(), msg.size());
    }

}
```

Stupid example: Let's make it parallel !

```
int main(int argc, char** argv){

    // creat socket
    socket_desc = socket(AF_INET ,
       SOCK_STREAM , 0);

    // bind the socket
    bind(socket_desc,  server , sizeof(server));
    listen(socket_desc , 100);

    //accept connection from an incoming client
    while(1){
           // declarations

           client_sock = accept(socket_desc,
 (struct sockaddr *)&client, &c);
           new std::thread(bind(do_work, client_sock));
       }
}
```

```
void do_work(int socket){

           //Receive a message
           while( (read_size =
                      recv(client_sock ,
client_message , 2000 , 0)) > 0{
              // Wonderful, we have a clie
              // useful works
           }
}
```

Wonderful and easy isn't it ?

Wonderful and easy isn't it ?

➡ **It does NOT scale**

Why this does not scale ?

```
void do_work(int socket){

        //Receive a message
        while( (read_size =
                recv(client_sock ,
client_message , 2000 , 0)) > 0){
          // Wonderful, we have a client
          // useful works
        }
}
```

→ Blocking IO

→ Your thread will spend most of the time to wait in I/O

→ Limiting factor : number of threads you can spawn

Solution ?

Use asynchronous I/O and event based model

Solution : Event based

Reactor Pattern and NON blocking I/O

1- One event loop for incoming I/O events

→ Use event monitoring function

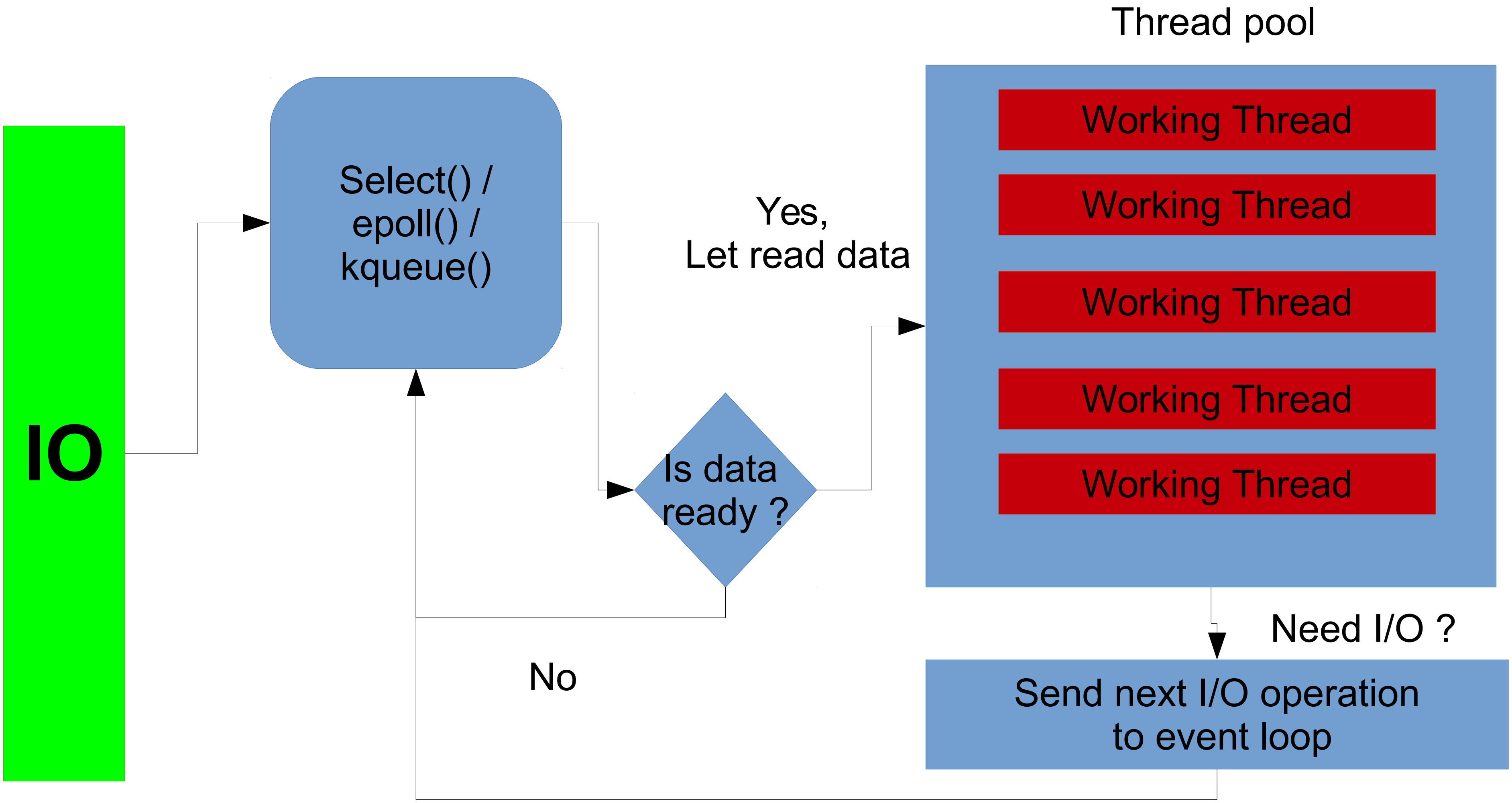
→ Select()/ poll() / epoll()/ kqueue()

2- The events are dispatched into tasks

3- Execute tasks in a ThreadPool

4- Send back I/O operations to themain thread

Event I/O architecture



Advantages of Reactor pattern

→ No need to spawn one thread per query
  - → Thread pool for task execution
  - → Lower memory consumption

→ Keep thread doing active work
  - → Maximize processor usage

→ Allow for fine grain scheduling with requests

You can use existing solutions

→ Boost Asynchronous I/O : ASIO

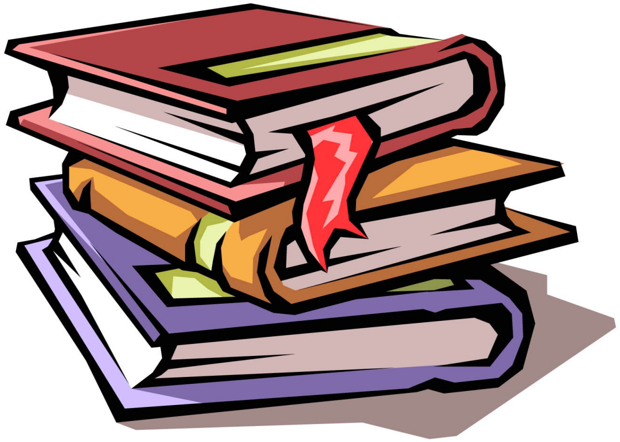
→ Libevent (C)
  → Most mature implementation

→ LibUV
  → node.js backend

→ POSIX ASIO asynchronous I/O
  → scalability limited

→ Green Threads
  → If your language support it

More about this



References :

- C10K publication :
  - http://www.kegel.com/c10k.html
- Boost ASIO documentation examples :
  - http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/examples/cpp11_examples.html
- LibEvent website:
  - http://libevent.org/
- Reactor vs proactor pattern
- Node.js

Conclusion

- Use asynchronous I/O in I/O bound softwares
- Use a ThreadPool instead of One thread per request
- Use task/event base model.