Asynchronous Design Pattern for Many-Core and Cloud Operating Systems

Bruno Santos
IT - UA
<bsantos@av.it.pt>

João Paulo Barraca
DET - UA
<jpbarraca@av.it.pt>

Rui L. Aguiar
DET - UA
<ruilaa@ua.pt>

Introduction

One of the key aspects for many-core and cloud operating systems is message passing communication. The networking aspect of this mechanism requires a high degree of concurrency to handle communication with thousands of cores simultaneously in an efficient and scalable manner. Non-blocking interfaces allows us to efficiently use a core and gives us concurrency without multiple threads, this also minimizes context switch.

We developed an asynchronous design pattern for operating systems written in C++. The framework uses only C++ language features (template metaprogramming), thus not requiring a custom compiler. The pattern provides a framework to develop and use asynchronous interfaces based on completion handlers. As the name implies, a completion handler is executed when the operation completes. The framework also provides queuing mechanism to serialize execution of handlers for synchronization purposes.

Framework

```
auto_buffer buff(4096);
size_t rbytes = read(buff);
//handle read completion
```

Code 1: Synchronous call

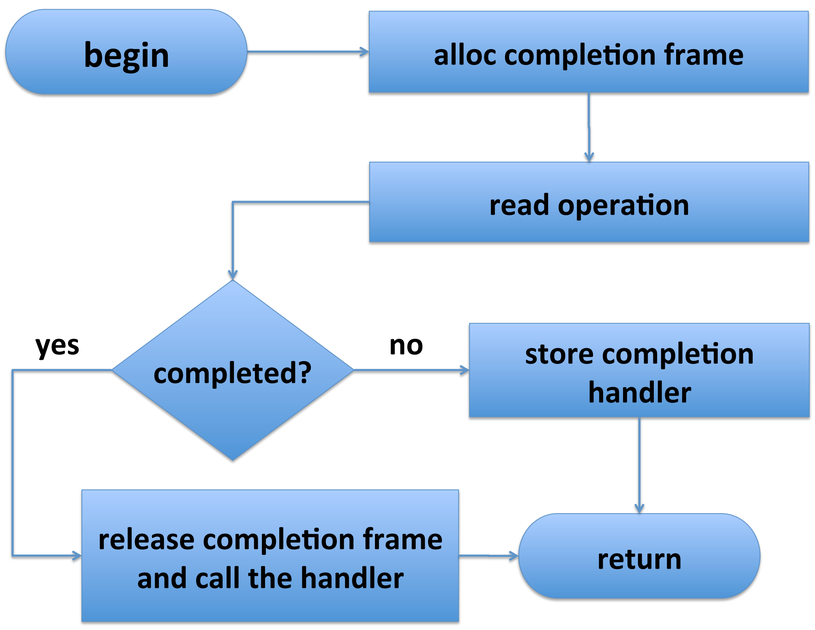
```
auto_buffer buff(4096);
read(buff, [buff](size_t rbytes) {
    //handle read completion
});
```

Code 2: Asynchronous call

```
template<class CompletionHandler>
inline void read(auto_buffer buff, CompletionHandler handler)
{
    async& as = get_async();
    auto frame = as.prolog<CompletionHandler, size_t>();

    do_read(buffer, get<0>(frame));
    as.postlog(handler, frame);
}
```

Code 3: Asynchronous call wrapper using our framework



Flowchart 1: Asynchronous call wrapper

```
void do_read(auto_buffer buff, tuple<size_t>& result)
{
    //request read operation

    if (needs to wait) {
        async& as = get_async();

        operation->completion = address_of(as);
        as.wait([&result, operation]() {
            get<0>(result) = operation->bytes_read;
        });
    } else {
        get<0>(result) = operation->bytes_read;
    }
}
```

Code 4: Asynchronous call implementation using our framework

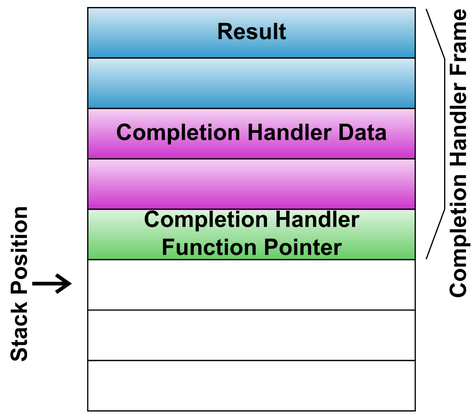
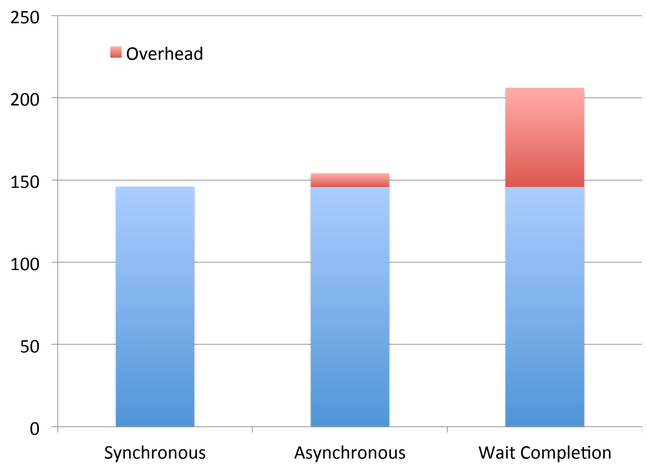
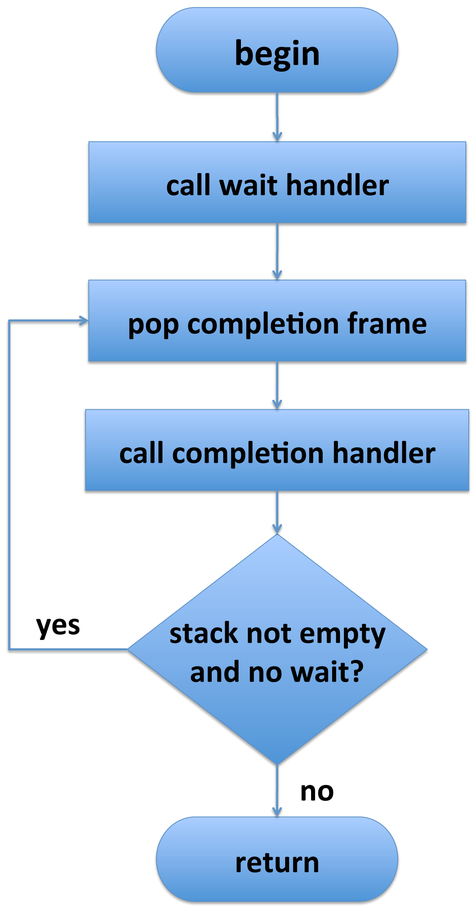


Figure 1: Async primitive stack



Graphic 1: Results in clock cycles



Flowchart 2: Wait completion

Conclusion

The presented framework enables writing and using asynchronous interfaces with little effort and without resorting to events or worker threads. The model's simplicity and negligible overhead allows us to write highly scalable message passing designs in a operating system for many-cores and clouds.

instituto de telecomunicações

universidade de aveiro

ATNoG

Project acronym: S(o)OS
Project full title: Service-oriented Operating Systems
Grant agreement no.: 248465

SEVENTH FRAMEWORK PROGRAMME