Event-driven Programming

.. as opposed to procedural decomposition

Karel Kubat / karel@kubat.nl / 2014

Contents

- Procedural decomposition vs. event-based design
- Generic examples of event callbacks
- Parallelism: forking vs. threading
- Node.js, libevent2, C++ Poco, your own event framework

- http://www.kubat.nl/pages/lectures (notes + sources)
- karel@kubat.nl

Decomposition (1)

Typical paradigm

- Analyze problem
- Decompose to subtasks
- Use iteration / recursiveness
- If necessary, use parallel processing

Decomposition (2)

```
doTask_a() {
   doSubTask_a1();
   doSubTask_a2();
   doSubTask_a3();
}
```

```
main() {
   doTask_a();
   doTask_b();
   doTask_c();
}
```

Event-based Design

- Select appropriate framework
- Prepare tasks as event handlers
- Register your event handlers
- Give control to framework
- Lean back
- Event handlers will be called when appropriate

Event handler example: C SAX parser

```
extern void xmlstardocument(void *ctx), xmlenddocument(void *ctx),
            xmlstartElement(void *ctx, char const *tag), ...
xmlSAXHandler myhandler;

myhandler.startDocument = xmlstartdocument;
myhandler.endDocument = xmlenddocument;
myhandler.startElement = xmlstartelement;
...

xmlSAXUserParseMemory(&myhandler, 0, buffer, strlen(buffer));
```

Event handler example: Perl SAX

```
my $parser = new XML::Parser(Handlers => {
    Init =>      \&init,
    Start =>     \&start,
    End =>       \&end,
    ... });
sub init { ... }
sub start { ... }
sub end { ... }

$parser->parsefile("test.xml");
```

Callbacks: XForms

```
int main(int argc, char **argv) {     /* Linux Journal                        */
    FL_FORM *form;                     /* Volume 1996, issue 22es (feb)       */
    FL_OBJECT *obj;                    /* XForms Tutorial and Review          */
                                       /* Karel Kubat, RijksUniv. Groningen */
    fl_initialize(&argc, argv, 0, 0, 0);

    form = fl_bgn_form(FL_UP_BOX, 230, 180);
    obj = fl_add_button(FL_NORMAL_BUTTON, 20, 20, 190, 60, "Hello world");
    fl_set_object_callback(obj, button_cb, 0);
    fl_end_form();

    fl_show_form(form, FL_PLACE_MOUSE, FL_FULLBORDER, "Hello, world!");
    fl_do_forms();

    return 0;
}
```

Asynchronicity: parallel processing

- Forking
  - Typical old style Unix approach
  - Safe, resource-intensive
- Threads
  - Originates from Windows
  - Fast

Asynchronicity: forking

```
doTask_a() {
   if (! (pid = fork()) )
      doTask_a1();
   if (! (pid = fork()) )
      doTask_a2();
   if (! (pid = fork()) )
      doTask_a3();
}
```

Forking - pro's and cons

Separate program space

- Difficult IPC (shared memory / socket IO)
- Child processes exit after completion
- When child crashes, parent doesn't
- When child doesn't free, it's ok
- Full use of exceptions & abnormal terminations

Asynchronicity: threads

```
extern doTask_a(), doTask_b();

 main() {
   id1 = pthread_create(doTask_a);
   id1 = pthread_create(doTask_b);
   pthread_join(id1); pthread_join(id2);
}
```

Threads - pro’s and cons

Identical program space

- Multiple parallel execution points
- Easier inter-thread communication, but use mutexes
- Each thread crash is fatal for all
- Thread context sanity: NO EXCEPTIONS!

Callbacks: node.js (1)

```
function my_callback(err, fd){
    if (err)
        // handle error
    else
        // fd is readable
}
fs.open('/tmp/myfile', 'r', my_callback);
console.log('Hello World');
// my_callback and console.log run at the same time!
```

Callbacks: node.js (2)

Anonymous function:

```
fs.open('/tmp/myfile', 'r', function() {
    . . .
});

console.log('Hello World!');
```

Callbacks: node.js (3) (cp1.js)

```
fs.open('c1.c', 'r', function(err, fdin) {
    if (err)
        util.error('Cannot read c1.c');
    else
        fs.open('c1.c.copy', 'w', function(err, fdout) {
            if (err)
                util.error('Cannot write c1.c.copy');
            else {
                // fdin is readable, fdout is writable
            }
        ));
});
```

Callbacks: node.js (4) (cp1.js)

```
function cp(src, dst, callback) {
    fs.open(src, 'r', function(read_err, readfd) {
      fs.open(dst, 'w', function(write_err, writefd) {
          // Even sync'd copy readfd => writefd here
          // is an improvement!
      });
    });
}
cp('a', 'b', function() { ... });
cp('c', 'd', function() { ... });
```

Node.js: Ad-hoc servers & clients

- Typical use: ad-hoc (small) network servers & clients
- Asynchronous client serving
- Example: files lister via http
  - http://server:port/ = directory listing
  - http://server:port/0 = show 1st file, /1 = second etc.

Node.js: file lister (1)

```
// Named server object, named callback
function callback(request, response) { ... }
var server = http.createServer(callback);
server.listen(8888);

// Anonymous server object, anonymous callback
http.createserver(function(req, resp) {
    ...
}).listen(8888);
```

Node.js: file lister (2) (s1.js)

```
http.createserver(function(req, resp) {
    // List current directory
    // If requested URL is slash:
        // Return directory listing
    // Else, we expect /0, /1, /2 etc.
        // Check that number is in range
        // Return contents of that file
}).listen(8888);
```

libevent2: Networking in C (1)

- Example: echo server
- main() creates listener: bound to IP/port, 2 callbacks:
  - acceptcb() callback (when client's connection is accepted)
  - errorcb() callback for errors
- acceptcb() creates I/O buffer

libevent2: Networking in C (2) (c4.c)

- acceptcb(): creates I/O buffer for TCP socket I/O
- Callbacks:
  - On readable - readcb()
  - On written - none
  - On statechange event - statechangecb()
- readcb(): copies read data to output buffer
- statechangecb(): frees resources at end

Event callbacks in C++ (& Java...)

Paradigm:

- Pure virtual class with event handler, state reporter, error handler, etc.
- If instantiation is deeper in the code: provide a factory that creates non-virtual objects

Poco: Networking in C++ (1)

```
class MyRequestHandler: public HTTPRequestHandler {
     // Specific handlers for fulfilling an HTTP request
}
class MyRequestHandlerFactory: public HTTPRequestHandlerFactory {
     HTTPRequestHandler *createRequestHandler(HTTPRequest const &req) {
          return new MyRequestHandler;
     };
}
class MyServer: public ServerApplication {
     int main(vector<string> args) override {
          HTTPServer(new MyRequestHandlerFactory, ServerSocket(9090), ...);
     }
};
```

Poco: HTTP hit counter (1)

```
// Class
class MyRequestHandler: public HTTPRequestHandler {
public:
    virtual void handleRequest(HTTPRequest &req,
                               HTTPResponse &resp);
private:
    static int count;
};
```

Poco: HTTP hit counter (2) (c5.cc)

```
// Implementation
int MyRequestHandler::count = 0;

void MyRequestHandler::handleRequest(HTTPRequest &req,
                                     HTTPResponse &resp) {
    resp.setStatus(HTTPResponse::HTTP_OK);
    resp.setContentType("text/html");
    ostream &out = resp.send();
    out << "<html><body>" << ++count << "</body></html>";
    out.flush();
}
```

Roll your own framework

My Own Framework

1. Define what you need
2. Build framework fit-to-match
3. Profit

Mof (1): Event Task template

```
class EvTask {
public:
    virtual ~EvTask();
    virtual bool ready() = 0;
    virtual void handle() = 0;
};
```

Mof (2): Event Scheduler

```
class EvScheduler {
public:
    void addtask(EvTask *task);
    void loop();
};
```

Mof (3): main()

```
int main() {
  EvScheduler scheduler;

  scheduler.add(new Counter(0, 40));
  scheduler.add(new Counter(60, 70));
  scheduler.add(new FileCopier("BigFile", "BigFile.
copy"));
  scheduler.add(new WebClient("www.kubat.nl"));
  scheduler.loop();

  return 0;
```

Mof (4): counter (c6.cc)

```
class Counter: public EvTask {
public:
    Counter (int min, int max): _min(min), _max(max),
                                _curr(min) {}
    bool ready()  { return _curr >= _max; }
    void handle() { cout << "Now at " << _curr++ << endl; }
private:
    int _min, _max, _curr;
};
```