event-oriented programming

UNICAMP — April 22, 2009

Gustavo Sverzut Barbieri <barbieri@profusion.mobi>

ProFUSION embedded systems

agenda

- introduction
- main loop
- unix: fds and poll/select
- hands on: gui

introduction

event-oriented programming

- reactive
- sleeps most of the time
- work is usually short/quick
- lives forever (until requested to quit)

solvable problems

- hardware (sensors) events
- user interface
- client/servers
- ... basically every non-batch!

ProFUSION embedded systems

using multiple threads

- one thread per resource (client, sensor...)
- common resources needs locking
- underlying libraries must cooperate
- operating system segments load for you

using single thread

- one thread to rule them all
- no need to lock resources
- no special needs on underlying libraries
- need to segment load for you

single versus multiple threads: rule

general rule:

*multiple is just good when work load is **hard to segment**.*

for everything else use single thread instead

multiple threads: examples

- non-snapshotable calculations
- blocking calls (includes syscalls)

sending static files in webserver can be done with `sendfile()`, it's better to run this from threads!

ProFUSION embedded systems

single thread: examples

- graphical user applications
- handling non-thread safe dbs (sqlite)
- non-blocking calls (includes syscalls)

interacting with various web2.0 services (soap/xmlrpc) is much easier from single threads!

using both single and multiple threads

they are not exclusive concepts!

single thread based application can start threads to do some work, then communicate to the “main” thread using standard communication primitives.

multiple threads based application can run single thread sub case in one of its threads.

always pay attention to resource sharing!

cooperative threads

- used on single thread applications
- segments work load
- share concepts with distributed computing
- handled as a pair **(function, context)**
  - **function:** what to execute, depends on context
  - **context:** state information, data, etc
- also implemented as coroutines

cooperative threads: gui-db example (1/2)

traditional non-cooperative (blocking) example:

```
function load(gui, query):
    while not query.is_last():
        row = query.next_row()
        gui.append(row)
```

blocks for a period dependent on number of elements

cooperative threads: gui-db example (2/2)

cooperative example:

```
function load(gui, query):
    if query.is_last():
        return stop
    row = query.next_row()
    gui.append(row)
    return continue
```

still blocks! but constant time independent on number of elements

cooperative threads: net-calcs example (1/2)

```
function calc(start, end, client):
    result = 0
    for i from start to end:
        result += part_calc(i)
    client.send(result)
```

ProFUSION embedded systems

cooperative threads: net-calcs example (2/2)

```
function calc(ctx, client):
    last = ctx.base + ctx.step
    if last > ctx.end:
        last = ctx.end
    for i from ctx.base to last:
        ctx.result += part_calc(i)
    if last == ctx.end:
        client.send(ctx.result)
        return stop
    ctx.base = last + 1
    return continue
```

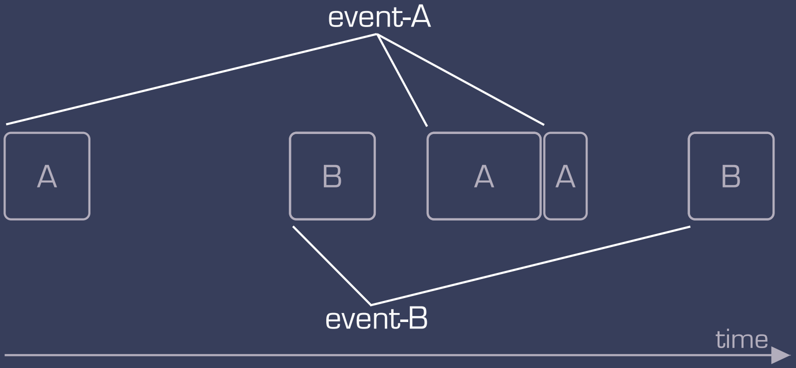
main loop

main loop: simplified

```
while (1) {
    wait();
    process();
}
```

ProFUSION embedded systems

main loop sequence

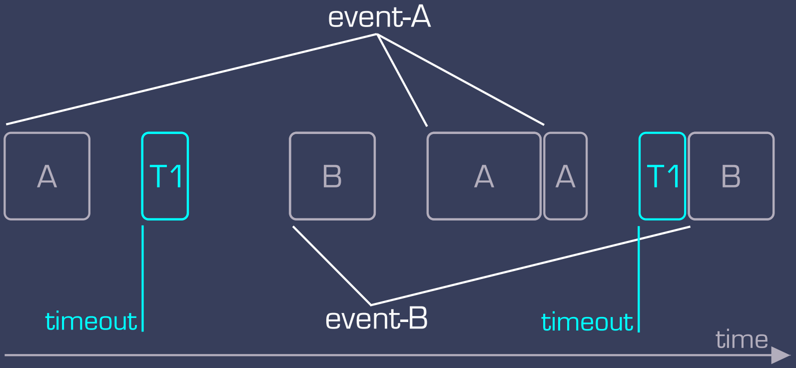


main loop: more real

```
while (1) {
    maxtime = first_expire();
    timeout = wait(maxtime);
    if (timeout && timers)
        process_timers();
    if (!timeout)
        process();
}
```

does not enable cooperative threads!

main loop sequence with timers

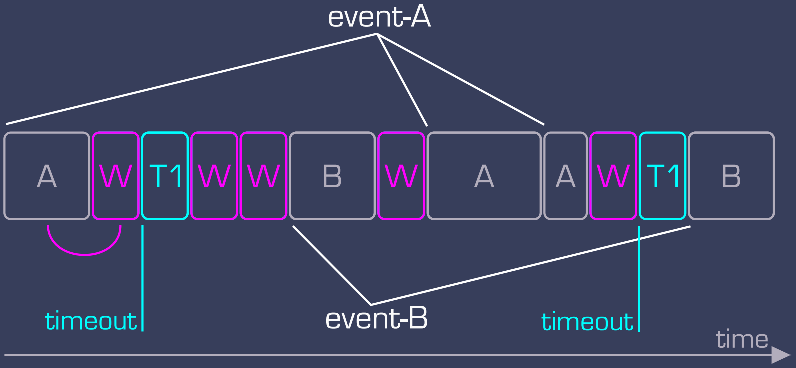


main loop: likely real

```
while (1) {
    maxtime = first_expire();
    timeout = wait(maxtime);
    if (timeout && timers)
        process_timers();
    if (!timeout)
        process();
    process_idlers();
}
```

does enable cooperative threads!

main loop sequence with timers



unix: file descriptors and poll/select

file descriptors

- originally abstract key for accessing a file
- expanded to cover sockets, directories, fifos...
- fancy and low level controls (`ioctl`, `fcntl`...)
- some can be mapped to process memory (*mmap*)
- can be read, written ... and **monitored!**

things that are file descriptors

- **files**
- directories
- character devices (modem)
- block devices (disk)
- **network sockets** (ip, tcp, udp, bluetooth...)
- **fifos** (named pipes)
- **pipes**
- even timers (`timerfd()`)
- and general events! (`eventfd()`)

monitoring file descriptors

- can I read from it without blocking?
- can I write to it without blocking?
- did errors occurred? (connection closed, ...)

note: read/write operations refer to basic units, usually a **byte!** doing more than that can still block if file descriptor is in blocking operation.

monitoring file descriptor the unix way

family of functions to monitor set of file descriptors for action or return on timeout:

- `select()`, original call to monitor file descriptors, painful to use. Uses bitmask and thus has fixed size/limit on number of file descriptors.
- `poll()` easy to use call, uses an array so no imposed limit.
- `epoll_wait()` new call to allow higher level of control (edge or level triggered events).

ProFUSION embedded systems

real world file descriptor usage: httpd

browse the web is all about file descriptors:

- servers (apache) creates one socket and `select()`
- when ready servers `accept()`
- `accept()` returns direct fd to client
- clients (browsers) `connect()` using sockets
- servers use other fds to read from files
- clients use other fds to cache to files

httpds usually mix threads and fork to handle clients after `accept()`, some use `select()`.

real world file descriptor usage: dbus

the external music player panel that lives on desktop to control amarok/rhythmbox:

- dbus daemon create a unix socket and `select()`
- music player `connect()` and registers a name
- music player toolkit (qt/gtk...) `select()`
- panel `connect()` and asks for music player
- panel toolkit `select()`
- music player signals are caught on panel's `select()`
- panel calls are caught on music player's `select()`
- toolkits process dbus and dispatch user calls

real world file descriptor usage: gui/x11

x11 is a client-server system. `server` is the one that connect to devices like vga, keyboard and mouse. `client` is usually the application.

- server open devices (vga, keyboard, mouse)
- server will create unix/tcp sockets
- server `select()`
- client `connect()` to server
- server wakes from `select()` on mouse and `write()` to client
- client wakes from `select()` and updates, `write()` to server

conclusion

conclusion

- event oriented programming is used a lot
- easy to integrate using main loop
- main loops can save you from thread hell
- need to take care when segmenting load
- poor segmented loads can make it sluggish

hands on: gui development

`strace:` tool to trace system calls and signals.

thanks!

Gustavo Sverzut Barbieri

barbieri@profusion.mobi
http://blog.gustavobarbieri.com.br/
http://profusion.mobi/

ProFUSION embedded systems