Exploring Linux API

Practical Asynchronous and Interprocess Communication Patterns

Henrique Marks

May 15, 2025

TOC

1. Overview
2. Low-Level Application
3. Middle Level Application
4. Ethernet Receiver - Test Application
5. Low-Level Application - Multiplexing
6. Low-Level Application - Signals ad Timers
7. Integration with External Libraries

Introduction

Topics

- Take a tour of some Linux User space APIs.
- Focus on Event-Driven Programming.
- Focus on Inter-Process Communication mechanisms.
- Integration with other libraries.

Introduction

Topics

- Take a tour of some Linux User space APIs.
- Focus on Event-Driven Programming.
- Focus on Inter-Process Communication mechanisms.
- Integration with other libraries.

Example Driven

- Create a multi-application project.
- For each application, show some IPC mechanism and the event-driven main loop.
- Add more and more functionality.

Project

Turtle!

- A device can be moved forward (1 step) or rotated (90°). It can be controlled via Ethernet.
- Create a Low-Level Application that sends Ethernet commands to the device.
- Create a middle-level application that communicates with the low-level app, sending the device to some position.
- Create a high-level application(GUI) that communicates with the middle-level one.
- Additional: testing apps.

Project

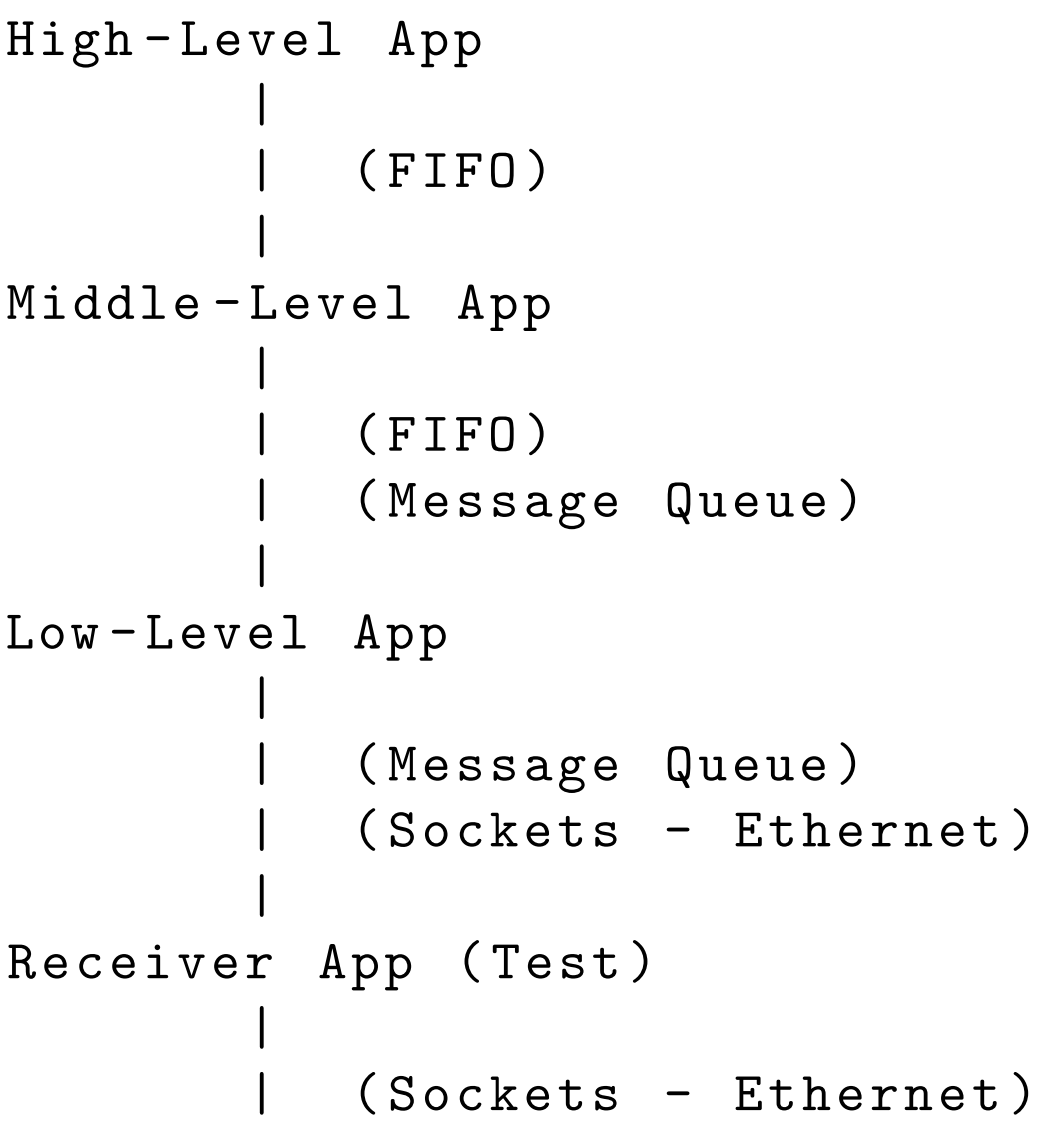
Turtle!

- A device can be moved forward (1 step) or rotated (90°). It can be controlled via Ethernet.
- Create a Low-Level Application that sends Ethernet commands to the device.
- Create a middle-level application that communicates with the low-level app, sending the device to some position.
- Create a high-level application(GUI) that communicates with the middle-level one.
- Additional: testing apps.

Definition

- All applications are event-driven.
- Showing different IPC mechanisms.

Architecture



Low-Level Application - main

Low-Level main application

- Create the LowLevelApp object, pass the message queue name and the Ethernet interface name
- Run the event loop.

```
int main()
{
    LowLevelApp app("/turtle_cmd", "tap0");
    app.run();
}
```

Low-Level Application - The API

Design Choices and API

- Low Level Application knows about moving Forward and Rotating.
- Translates the message queue commands into Ethernet.
- It is stateless.

```
enum class CommandType : uint8_t
{
    MOVE_FORWARD = 1,
    ROTATE_90 = 2
};
```

Low-Level Application - The API

Design Choices and API

- Low Level Application knows about moving Forward and Rotating.
- Translates the message queue commands into Ethernet.
- It is stateless.

```
enum class CommandType : uint8_t
{
    MOVE_FORWARD = 1,
    ROTATE_90 = 2
};
```

Design

- This presentation does not focus on design.
- But a better design should hide this enum in a proper API.

Low-Level Application - Definition

```
class LowLevelApp {
public:
    LowLevelApp(string& mq_name, string& interface);
    void run();

private:
    int mq_fd_;
    int raw_sock_fd_;
    int epoll_fd_;
    std::string interface_;

    void setup_message_queue(string& mq_name);
    void setup_raw_socket();
    void event_loop();
    void handle_command();
    void send_ethernet_command(CommandType cmd);
};
```

Low-Level Application - constructor

```
LowLevelApp::LowLevelApp(string& mq_name, string&
    interface) : interface_(interface) {

    setup_message_queue(mq_name);
    setup_raw_socket();

    epoll_fd_ = epoll_create1(0);

    epoll_event ev{};
    ev.events = EPOLLIN;
    ev.data.fd = mq_fd_;
    epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, mq_fd_, &ev);
}
```

Low-Level Application - setup_raw_socket

```
void LowLevelApp::setup_raw_socket() {
  raw_sock_fd_ = socket(AF_PACKET, SOCK_RAW, htons(0
      x88B5));

  struct ifreq ifr{};
  std::strncpy(ifr.ifr_name, interface_.c_str(),
      IFNAMSIZ);

  sockaddr_ll saddr{};
  saddr.sll_family = AF_PACKET;
  saddr.sll_ifindex = ifr.ifr_ifindex;
  saddr.sll_protocol = htons(0x88B5);

  bind(raw_sock_fd_, reinterpret_cast<sockaddr*>(&
      saddr), sizeof(saddr));
}
```

Low-Level Application - setup_message_queue

```
void LowLevelApp::setup_message_queue(string& mq_name)
{
    struct mq_attr attr{};
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = 1;

    mq_fd_ = mq_open(mq_name.c_str(), O_RDONLY |
        O_CREAT, 0644, &attr);
}
```

Low-Level Application - event_loop

```
void LowLevelApp::event_loop() {
    while (true) {
        epoll_event ev{};
        int nfds = epoll_wait(epoll_fd_, &ev, 1, -1);
        if (nfds > 0 && ev.data.fd == mq_fd_) {
            handle_command();
        }
    }
}
```

Low-Level Application - handle_command

```
void LowLevelApp::handle_command()
{
 char buffer;
 ssize_t n = mq_receive(mq_fd_, &buffer, 1, nullptr);
 if (n == 1)
 {
  CommandType cmd = static_cast<CommandType>(buffer);
  send_ethernet_command(cmd);
 }
}
```

Send Ethernet Frame

Sending the Ethernet Frame is not going to be shown here, refer to the repository. It is just sent using the socket and bit banging.

Middle Level Application

Explanation

- The middle level app receives commands from a FIFO.
- The commands are simple: "move to position (x,y)".
- It implements an algorithm to do the movement.
- It translates to the low level interface (commands in the message queue).
- It is state-full, keeping the position of the object.

Middle Level Application

Explanation

- The middle level app receives commands from a FIFO.
- The commands are simple: "move to position (x,y)".
- It implements an algorithm to do the movement.
- It translates to the low level interface (commands in the message queue).
- It is state-full, keeping the position of the object.

Differences

- It is the same type of application. Event-Driven, epoll based, but using a FIFO, and writing to the message queue.
- Let's check the differences.

Middle-Level App - create and handle FIFO

```
void createFifo() {
    mkfifo(FIFO_PATH, 0666);
    fifo_fd_ = open(FIFO_PATH, O_RDONLY | O_NONBLOCK);
}

void handleFifoInput() {
  char buf[256] = {0};
  ssize_t count = read(fifo_fd_, buf, sizeof(buf)-1);
  std::istringstream iss(std::string(buf, count));
  // Read Line, parse and call algorithm() method
  // Threads and State Machines here
}
```

Middle-Level App - send_command

Algorithm and Commands

Given the initial position and the requested position, an algorithm defines the sequence of FORWARD and ROTATE commands to send to the low-level app.

```
void sendCommand(uint8_t cmd) {
    mq_send(mq_, reinterpret_cast<const char*>(&cmd),
        1, 0);
}
```

Ethernet Receiver - Test Application

Explanation

- The receiver application is simple, just receive Ethernet Packets.
- Because of that, let's use io_uring instead of epoll!

Ethernet Receiver - Test Application

Explanation

- The receiver application is simple, just receive Ethernet Packets.
- Because of that, let's use io_uring instead of epoll!

Differences

- io_uring is newer, from kernel 5.1, and later on adopted in Userspace apps. It has been created to be used when there is a need for greater throughput, and as always, low-latency. It can be implemented without resorting to receiving threads (done by kernel queues).

Ethernet Receiver - io_uring init

```
io_uring ring;
io_uring_queue_init(QUEUE_DEPTH, &ring, 0) != 0);
```

Ethernet Receiver - io_uring init

```
io_uring ring;
io_uring_queue_init(QUEUE_DEPTH, &ring, 0) != 0);
```

Explanation

- Initializes an io_uring instance with a submission/completion queue of size QUEUE_DEPTH.
- ring: Holds internal state for io_uring operations.
- QUEUE_DEPTH: Number of concurrent I/O operations we will queue (here, 8).

Ethernet Receiver - Submission Queue

```
io_uring_sqe* sqe;
sqe = io_uring_get_sqe(&ring);
io_uring_prep_recv(sqe, sock_fd, buffers[i],
    BUFFER_SIZE, 0);
io_uring_sqe_set_data(sqe, buffers[i]);
io_uring_submit(&ring);
```

Ethernet Receiver - Submission Queue

```
io_uring_sqe* sqe;
sqe = io_uring_get_sqe(&ring);
io_uring_prep_recv(sqe, sock_fd, buffers[i],
    BUFFER_SIZE, 0);
io_uring_sqe_set_data(sqe, buffers[i]);
io_uring_submit(&ring);
```

Explanation

- Declare an associate submission queue to the ring.
- Associate a recv syscall to the sqe, passing the fd and buffers.
- Buffers will be returned by sqe.
- Submit the ring and sqe to the kernel.

Ethernet Receiver - Completion Queue

```
while (true) {
    io_uring_cqe* cqe;
    io_uring_wait_cqe(&ring, &cqe);
    uint8_t* data = io_uring_cqe_get_data(cqe);
    ssize_t len = cqe->res;
    /* Use data */
    io_uring_cqe_seen(&ring, cqe);
}
```

Ethernet Receiver - Completion Queue

```
while (true) {
   io_uring_cqe* cqe;
   io_uring_wait_cqe(&ring, &cqe);
   uint8_t* data = io_uring_cqe_get_data(cqe);
   ssize_t len = cqe->res;
   /* Use data */
   io_uring_cqe_seen(&ring, cqe);
}
```

Explanation

- Declare and wait on a completion queue related to the ring.
- Get data and length.
- Mark the Completion Queue as seen.
- 2 queues: sqe writes buffer to cqe.

Low-Level App - Add Logs

Explanation

- Application uses a small log library.
- Log Level is configured via configuration file.
- Monitor file change, and call hlog_reload method to update log level.
- Use inotify API to monitor file changes
- Epoll loop will multiplex, monitor multiple file descriptors.

Logs - setup_inotify

```
void LowLevelApp::setup_inotify() {
  inotifyFd_ = inotify_init1(IN_NONBLOCK);
  string path = "hlog.conf";
  inotifyWatch_ = inotify_add_watch(inotifyFd_, path.
      c_str(), IN_CLOSE_WRITE);
}
```

Logs - handle_inotify

```
void LowLevelApp::handle_inotify()
{
    struct inotify_event *event;
    ssize_t len = read(inotifyFd_, buf, sizeof(buf));

    for (char *ptr = buf; ptr < buf + len; ptr +=
        sizeof(struct inotify_event) + event->len)
    {
        event = (const struct inotify_event *)ptr;
        if (event->mask & IN_CLOSE_WRITE)
        {
            HLOG_INFO("Reload Called");
            hlog_reload();
        }
    }
}
```

Logs - epoll_loop

```
void LowLevelApp::event_loop() {
  const int MAX_EVENTS = 10;
  struct epoll_event ev[MAX_EVENTS];
  while (true) {
    int nfds = epoll_wait(epoll_fd_, ev, MAX_EVENTS,
        -1);
    for (int i = 0; i < nfds; ++i) {
      if (ev[i].data.fd == mq_fd_) {
        handle_command();
      }
      else if (ev[i].data.fd == inotifyFd_) {
        handle_inotify();
      }
    }
  }
}
```

Low-Level App - Signal and Timer

Signal

- Use Ctrl-C to quit the application
- Capture the signal SIGINT, and print a nice message
- Use SignalFD API, to integrate nicely with epoll.

Timer

- Every 10 seconds, simulate that we send a "keep-alive message"
- Use TimerFD API, to create the timer and integrate nicely with epoll.

Signals - setup_signal

```
void LowLevelApp::setup_signal()
{
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigprocmask(SIG_BLOCK, &mask, nullptr);

    signal_fd_ = signalfd(-1, &mask, SFD_NONBLOCK);
}
```

Signals - handle_signal

```
void LowLevelApp::handle_signal()
{
    struct signalfd_siginfo si;
    ssize_t bytes = read(signal_fd_, &si, sizeof(si));

    if (si.ssi_signo == SIGINT)
    {
        std::cout << "\n[LowLevelApp] Caught SIGINT (
            Ctrl+C). Shutting down gracefully...\n";
        running_ = false;
    }
}
```

Timer - setup_timer

```
void LowLevelApp::setup_timer()
{
    timer_fd_ = timerfd_create(CLOCK_MONOTONIC,
        TFD_NONBLOCK);

    struct itimerspec ts{};
    ts.it_interval.tv_sec = 10;   // Repeating interval
    ts.it_interval.tv_nsec = 0;
    ts.it_value.tv_sec = 10;      // Initial expiration
    ts.it_value.tv_nsec = 0;

    timerfd_settime(timer_fd_, 0, &ts, nullptr);
}
```

Timer - handle_timer

```
void LowLevelApp::handle_timer()
{
  uint64_t expire;
  ssize_t s = read(timer_fd_, &expire, sizeof(expire))
      ;

  keepaliveNum_++;
  HLOG_DEBUG("[LowLevelApp] KeepAlive %d Sent",
      keepaliveNum_);
}
```

Integration with 0mq

0mq

- Use zmq_getsockopt(socket, ZMQ_FD, fd, fdSize) and epoll on fd.
- When triggered, call zmq_getsockopt(ZMQ_EVENTS) to check for read/write 0mq events.

Integration with MQTT - Mosquitto

Mosquitto API

- Create mosquitto client, connect and get fd
- int fd = mosquitto_socket(mosq);
- In the Epoll Loop use Mosquitto Non-Blocking API
- mosquitto_loop_read(mosq, 1);
- mosquitto_loop_write(mosq, 1);
- Call periodically the following method for keepalive messages
- mosquitto_loop_misc(mosq);

Conclusion and Links

Links

- 040coders May 2025 on GitLab
- Linux online man pages
- Nice book to have

Conclusion

Thanks!