Computer Labs: Event Driven Design

2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

October 23, 2017

Contents

Event Driven Design

State Machines

Event Handling

Events and I/O

Event An **event** is a change in the state

- Virtually all I/O processing is driven by events
  - Whether events are detected by interrupts or by polling
  - Even video graphics output may depend on events (synchronization with the vertical “movement” to avoid visual artifacts)
- Except for the labs on the video card, all others have been driven by I/O events
- Your project will also be event driven:
  - Its execution will depend on events generated by the I/O devices
    - Whether you use polling or interrupts for detecting these events.

Event Driven Design

- This is best addressed by an **event** driven design:
  - That is a design where **flow control** is determined by the **environment** rather than the program itself
  - Essentially, the code is executed **reactively** in response to events that may occur **asynchronously** with program execution
- Event driven design is particularly common in:
  - Graphical user interfaces (GUI)
  - Games
  - Communications/network software
  - Embedded systems

Simple Event Driven Design

Events The types of events that the different components of the system have to handle

Event Queues That provide the necessary buffering so that handling of an event may occur asynchronously to its occurrence

Event Handlers That process each type of event

Dispatchers That monitor the event queues and call the appropriate event handlers

- May be implemented as a simple loop that checks for events

Event Driven Design and Minix 3 DD Design

- We can find these elements of event driven design in the pattern used in the design of interrupt driven Minix 3 DDs

```
 5: while( 1 ) { /* You may want to use a different condition */
 6:     /* Get a request message. */
 7:     if ( driver_receive(ANY, &msg, &ipc_status) != 0 ) {
 8:         printf("driver_receive failed with: %d", r);
 9:         continue;
10:     }
11:     if (is_ipc_notify(ipc_status)) { /* received notification
12:         switch (_ENDPOINT_P(msg.m_source)) {
13:         case HARDWARE: /* hardware interrupt notification */
14:             if (msg.NOTIFY_ARG & irq_set) { /* subscribed int
15:                 ... /* process it */
16:             }
17:             break;
18:         default:
19:             break; /* no other notifications expected: do not
20:         }
21:     } else { /* received a standard message, not a notificati
```

Contents

Event Driven Design

State Machines

Event Handling

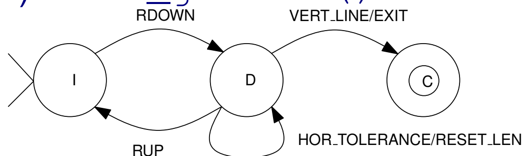
Event Driven Design and State Machines

- For other than simple designs, it is very helpful to use state machines in combination with event driven design
  - A state machine is useful to handle events that may depend on the state
- A state machine is itself event driven
  - The transition from one state to another depends on the occurrence of an event

Lab 4: `test_gesture()`

- The program should exit when the user "draws" a line with a positive slope, while pressing down the right button
- You can ignore some events, namely those related to other buttons.

Lab 4 (15/16): `test_gesture()` State Machine (1/2)



Cur. State	Input	Next State	Output
`I(nitial)`	`RDOWN`	`D(rawing)`	
`D(rawing)`	`VERT_LINE`	`C(omplete)`	`Exit`
`D(rawing)`	`RUP`	`I(nitial)`	
`D(rawing)`	`HOR_TOLERANCE`	`D(rawing)`	Reset length

`RDOWN` Right button has been pressed

`RUP` Right button has been released

`VERT_LINE` Vertical line with desired length drawn

- This is somewhat high-level
  - You need to detect the events from the packets received from the mouse
    - You can use other state machines

Lab 4 (15/16) `test_gesture()` - State Machine (2/2)

```
typedef enum {INIT, DRAW, COMP} state_t;
typedef enum {RDOW, RUP, MOVE} ev_type_t;
void check_hor_line(event_t *evt) {
     static state_t st = INIT; // initial state; keep state
     switch (st) {
     case INIT:
         if( evt->type == RDOWN )
             state = DRAW;
         break;
     case DRAW:
         if( evt->type == MOVE ) {
             [...] // need to check if events VERT_LINE  or HOR_TOLERANCE
         } else if( evt->type == RUP )
             state = INIT;
         break;
     default:
         break;
     }
}
```

- This is rather high-level
  - You need to add code to detect the events
    - This can be done with other state machines

(State Machines)

- This state machine is an example of a **Mealy Machine**, drawn “a la DFA”:
  - A state machine where the output depends **not only** on the state **but also** on the input event
- An alternative state machine is the **Moore Machine**:
  - A state machine where the output depends **only** on the state.
  - This usually leads to extra states
    - In this (simple) case the two machines are structurally equal
- They are essentially equivalent, but in an event-driven design the implementation of a Mealy machine is more straightforward:
  - For each state, when a **relevant event** occurs, just produce the output (if any) and change the state (this is a special output)
  - A **relevant event** may not be a raw event generated by an input device:
    - “Raw” events generated by mice are the reception of packet bytes, not the change in the state of mouse buttons

Contents

Event Driven Design

State Machines

Event Handling

Event Processing

- I/O devices' events are processed by the corresponding interrupt handlers
- The IHs may be

  Application Dependent For example, the mouse IH not only receives the mouse packets, but also detects the exit sequence

  Application Independent The mouse IH just receives the mouse packets. The exit sequence is detected by application dependent code.

    - Need to define an application dependent event handler
    - Need to specify how the IH "communicates" with this event handler
      - How the data received from the mouse is passed to the event handler?
      - When is the event handler executed?

Application Independent vs Application Dependent IH

In General

- Can be reused
  - Operating systems IH is independent of applications
- Introduces a level of indirection
  - May add flexibility
  - May be more responsive (IHs are shorter)
  - Requires more code (overall)
  - Has higher overhead

In Minix 3 Labs and Project

`driver_receive()` is a blocking call

- Application dependent processing must be done in the same iteration loop as application independent processing
  - It is not possible to delay application dependent processing until there are no interrupts to handle
- It does not afford as much flexibility as in the general case
  - This is **not** an issue for ordinary Minix 3 DDs

Minix 3 and Application Independent IHs

```
 5: while( 1 ) { /* You may want to use a different condition */
 6:     /* Get a request message. */
 7:     if ( driver_receive(ANY, &msg, &ipc_status) != 0 ) {
 8:         printf("driver_receive failed with: %d", r);
 9:         continue;
10:     }
11:     if (is_ipc_notify(ipc_status)) { /* received notification
12:         switch (_ENDPOINT_P(msg.m_source)) {
13:         case HARDWARE: /* hardware interrupt notification */
14:             if (msg.NOTIFY_ARG & irq_set) { /* subscribed int
15:                 ... /* process it */
16:             }
17:         ...
18:         }
19:     } else { /* received a standard message, not a notificati
20:     ...
21:     }
22:     /* Now, do application dependent event handling */
23:     if ( event & MOUSE_EVT ) {
24:         handle_mouse();
25:     } else if ( event &
```

Further Reading

- Máquinas de Estado em C