Concurrent Computing 6: Event-Driven Prog.

1. Introduction
2. Terminology
3. Synchronization
4. C and Concurrency
5. More Asynchrony
6. **Event-Driven Programming**

Event-Driven Programming

Event-driven (or **event-based**) **programming** is a *programming paradigm* (style) where program functionality is structured in terms of a set of **event handler** routines and **control flow** is determined by the sequence in which the relevant **events** occur.

Events can include a wide variety of things: keystrokes, mouse clicks, "messages" becoming available on a pipe/socket, changes in variable values, changes in the state of files in a directory, outputs from sensors, etc.

Event handlers are *subroutines/functions/methods* that perform the appropriate functionality whenever their associated event occurs.

Event-driven programming is widely used for **graphical user interfaces** (**GUI**s) as well as in **computer gaming** (multiple game AIs must respond to player and other AIs actions).

Event-Driven Programming (contd.)

The reason for discussing event-driven programming here is that it can be advantageous to use concurrent programming techniques to implement event-driven programs.

Conceptually, an event-driven program can be described as:

```
while (1) {
  //Wait for some event(s) to occur:
  while ((events = check_for_events()) == NULL);

  //If multiple events, decide which to handle next:
  event = select_event_to_handle(events);

  //Invoke the correct event handler for the selected event:
  dispatch_to_handler(event);
}
```

Event-Driven Programming (contd.)

The above code has the check for whether an event(s) has occurred inside a `while` loop that will repeat continuously until an event occurs.

While this is conceptually what happens in event-driven programming, a direct implementation of this sort of **busy waiting** or **polling** will be used only if absolutely necessary, since it wastes CPU cycles.

Instead, waiting/checking for an event to occur will typically be implemented using one or more the following OS capabilities:

- **I/O blocking**
- **I/O multiplexing** (using `select()`, `poll()`, etc.)
- **suspending** a process until a **signal** is delivered

Event-Driven Programming (contd.)

Perhaps the most obvious advantage of a concurrent program implementation of an event-driven program is that it could take advantage of a computer with multiple CPUs/cores by allowing event checking to continue to run in parallel with event handler calls.

This is particularly useful for maintaining fast *response time* when events may take signficant time to handle (limiting the time a handler hogs the CPU would be quite complicated in a single-thread program).

A straightforward way to add concurrency to the basic event-driven program logic would be by having each handler call get run in a new OS thread or process.

Event-Driven Programming (contd.)

However, since there is some *time cost* to creating new threads (especially new processes), it can make more sense to create a separate thread for each event handler when the program begins running, and simply *suspend* each thread until there is a relevant event that needs to be handled.

In fact, instead of continuing to have a central thread that is responsible for detecting events and doing handler dispatching, the logic for detecting the relevant event(s) can be distributed to each handler thread.

The event-driven program is now a set of threads each of which has the following logic:

```
while (1) {
  while ((event = check_for_handler_event()) == NULL);
  handler(event);
}
```

Event-Driven Programming (contd.)

As noted above, each thread will generally make use of OS capabilities to avoid busy-waiting for their event to occur.

In fact, doing so can be more critical with this design, since there would be multiple threads wasting CPU cycles, which could delay other threads from being scheduled to be run as soon as their event has occurred.