Announcements

- Program #4
  - Due on Tuesday
  - Reminder: We grade your *final* submission
  - See notes on web page regarding linking
- Reading
  - Notes (Today)
  - Notes (Tuesday)

Events

- Many programs need to respond to external activity
  - keyboard
  - mouse
  - network
  - sensors
- Many programs can expect input from multiple sources
  - often don't know where the next input will come
  - will the user press a key or move the mouse next?
- Events are a way to represent these activities
  - can also includes other things (such as timer alarms)

Event Driven Programming

- Program consists of a main loop waiting for events
- Define set of functions to handle each event
  - process event, and then return
- Often uses function pointers to define events and their handlers

Structure of an Event Driven Program

```
while (1) {
    event = getNextEvent();
    switch (event.type) {
        case MOUSE_EVENT:
            processMouse(event));
            break;
        case KEYBOARD_EVENT:
            processKeyboard(event);
            break;
        .....
    }
}
```

Handling Multiple Events - Select

- Often events are coming from file descriptors
  - keyboard input
  - file I/O
  - network communication
- Select system call
  - blocks waiting for I/O to be ready on a set of file descriptor
  - returns when one or more file descriptors have I/O ready
  - does **NOT** perform the I/O
  - can include a timeout
    - prevent blocking forever
    - allows time based events

Select System Call

- int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
  - n - highest numbered file descriptor in any set
  - readfds, writefds, exceptfds - sets of file descriptors
  - timeout - how long to wait for events
  - return
    - value is number of descriptors in sets
    - sets modified to indicate which file descriptors are active
- set routines
  - FD_ISSET(int fd, fd_set *set);
    - is fd in the set
  - FD_SET(int fd, fd_set *set);
    - added fd to set
  - FD_ZERO(fd_set *set);
    - clear out the set

Why select for writing?

- Why select for writing?
  - Output buffers have finite size.
    - All communication fd's have buffered output.
  - To keep devices busy while application does something else.
  - write() ordinarily blocks if buffer is full; waits for drain.
  - Easy for select() to tell if there is space.
  - So it makes sense to select() for writing.
- Why select for errors?
  - learn about I/O problems
  - learn when a network connection closes down

Select Example

```
int main() {
    int ret, myServerFD, maxFD = 0;    FD_SET set;
    myServerFD = connectToServer();
    while (1) {
        FD_ZERO(&set);
        FD_SET(0, &set);
        FD_SET(myServerFD, &set);
        if (myServerFD > maxFD) maxFd = myServerFD;
        ret = select(maxFd+1, &set, NULL, NULL, NULL);
        if (ret)  {
            if (FD_ISSET(0, &set)) { /* read from standard input */ }
            if (FD_ISSET(myServerFD, &set)) { /* read from net */ }
        }
    }
}
```

Handling I/O Events

- Reading from Standard input

```
void readStandardIO() {
    ret = read(0, line, sizeof(line));

    /* process line */

}
```

Coordinating Events

- Sometimes Events come from sources other than file descriptors
  - other threads
- Solution:
  - use a queue of events
  - threads wishing to communicate en-queue events
- main loop for event queue paradigm:

```
while (1) {
    event = myEventQueue->dequeue();
     /* process event */
    anotherEventQueue->enqueue(response);
}
```

How do event-driven programs maintain state?

- Ordinary programs, and threads, use variables on the stack.
  - Thread's stack persists across thread context switches
- Event-Driven programs can't keep state on the stack.
  - Since each callback must return immediately.
  - We want to associate data/state with each callback.
- Solution:
  - Keep data in heap allocated structure
  - associate structure instance with file descriptor
  - can include functionPointer+arg
    - function pointer defines what to call
    - arg defines state for that function

How to Keep Programs Modular

- Problem:
  - Multiple parts of the system want to process events
  - Everyone wants to write to a main loop
- Solution:
  - Define an interface to allow event handlers to register
  - Single common event loop
  - Callback functions to specific event handlers

Example Using Callbacks

```
typedef int (*handlerFunc)(int fd, void *arg);

typedef struct {
    handlerfunc handler;
    int fd;
    void *arg;
} eventHandler;
```