33. Event-based Concurrency

Operating System: Three Easy Pieces

Event-based Concurrency

- A different style of concurrent programming without threads
  - Used in *GUI-based applications*, some types of *internet server-side frameworks (v.gr. node.js)*.

- **The problem** that event-based concurrency addresses is two-fold.
  - Managing concurrency correctly in multi-threaded applications.
    - Missing locks, deadlock, and other nasty problems can arise.

  - The developer has little or no control over what is scheduled at a given moment in time.

The Basic Idea: An Event Loop

- The approach:
  - **Wait** for something (i.e., an "*event*")to occur.
  - When it does, **check** what type of event it is.
  - **Do** the small amount of work it requires.
- Example:

```
while(1){
     events = getEvents();
     for( e in events )
          processEvent(e); // event handler
}
```

A canonical event-based server (Pseudo code)

How exactly does an event-based server determine which events are taking place.

An Important API: `select()` (or `poll()`)

- Check whether there is any incoming I/O that should be attended to.
  - `select()`

Hint to the compiler
(about the scope of the pointer)

```
int select(int nfds,
           fd_set * restrict readfds,
           fd_set * restrict writefds,
           fd_set * restrict errorfds,
           struct timeval * restrict timeout);
```

- Lets a server determine that a **new packet has arrived** and is in need of processing.
- Let the service know when **it is OK to reply**.
- `timeout`
  - `NULL`: Cause `select()` to *block indefinitely* until some descriptor is ready.
  - `0`: Use the call to `select()` to *return immediately.*

Using `select()`

- How to use `select()` to see which network descriptors have incoming messages upon them.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    // open and set up a bunch of sockets (not shown)
    // main loop
    while (1) {
        // initialize the fd_set to all zero
        fd_set readFDs;
        FD_ZERO(&readFDs);

        // now set the bits for the descriptors
        // this server is interested in
        // (for simplicity, all of them from min to max)
        …
```

Simple Code using `select()`

Using `select()` (Cont.)

```c
            int fd;
            for (fd = minFD; fd < maxFD; fd++)
                FD_SET(fd, &readFDs);

            // do the select
            int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);

        // check which actually have data using FD_ISSET()
        int fd;
        for (fd = minFD; fd < maxFD; fd++)
            if (FD_ISSET(fd, &readFDs))
                processFD(fd);
        }
}
```

Simple Code using `select()` (Cont.)

Why Simpler? No Locks Needed

- The event-based server cannot be interrupted by another thread.
  - With a single CPU and an event-based application.
  - It is decidedly **single threaded**.
  - Thus, *concurrency bugs* common in threaded programs **do not manifest** in the basic event-based approach.

A Problem: Blocking System Calls

- What if an event requires that you issue **a system call** that might block?
  - There are no other threads to run: *just the main event loop*
  - The entire server will do just that: block until the call completes.
  - Huge potential waste of resources

In event-based systems: no blocking calls are allowed.

A Solution: Asynchronous I/O

- Enable an application to issue an I/O request and return control immediately to the caller, before the I/O has completed.
  - Example:

```
struct aiocb {
        int aio_fildes;              /* File descriptor */
        off_t aio_offset;            /* File offset */
        volatile void *aio_buf;      /* Location of buffer */
        size_t aio_nbytes;           /* Length of transfer */
};
```

    - An Interface provided on *Max OS X*
    - The APIs revolve around a basic structure, the `struct aiocb` or **AIO control block** in common terminology.
    - Standard API is defined by POSIX AIO
      - Linux, BSD, Solaris, …

A Solution: Asynchronous I/O (Cont.)

- Asynchronous API:
  - To issue an asynchronous read to a file

```
int aio_read(struct aiocb *aiocbp);
```

    - If successful, it returns right away and the application can continue with its work.

  - Checks whether the request referred to by `aiocbp` has completed.

```
int aio_error(const struct aiocb *aiocbp);
```

    - An application can **periodically pool** the system via `aio_error()`.
    - If it has completed, returns success.
    - If not, EINPROGRESS is returned.

A Solution: Asynchronous I/O (Cont.)

- Interrupt
  - Remedy the overhead to check whether an I/O has completed
  - Using **UNIX signals** to inform applications when an asynchronous I/O completes.
  - Removing the need to *repeatedly ask the system*: polling vs. interrupts

ASIDE: Unix Signals

- Provide a way to communicate with a process.
  - *HUP* (hang up), *INT*(interrupt), *SEGV*(segmentation violation), and etc.
  - **Example**: When your program encounters a *segmentation violation*, the OS sends it a *SIGSEGV*.

```
#include <stdio.h>
#include <signal.h>
void handle(int arg) {
        printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
        signal(SIGHUP, handle);
        while (1)
                ; // doin' nothin' except catchin' some sigs
        return 0;
}
```

A simple program that goes into an infinite loop

ASIDE: Unix Signals (Cont.)

- You can send signals to it with the **kill command** line tool.
  - Doing so will *interrupt the main while loop* in the program and run the handler code `handle()`.

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```

Another Problem: State Management

- The code of event-based approach is generally more complicated to write than *traditional thread-based* code.
  - It must package up some program state for the next event handler to use when the I/O completes.
  - The state the program needs is on the stack of the thread. → **manual stack management**

Another Problem: State Management (Cont.)

- **Example** (an event-based system) (read from disk and send to network):

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

  - First **issue** the read asynchronously.
  - Then, **periodically check** for completion of the read.
  - That call informs us that the **read is complete.**
  - How does the event-based server know what to do?

Another Problem: State Management (Cont.)

- **Solution:** continuation
  - **Record** the needed information to finish processing this event *in some data structure*.
  - When the event happens (i.e., when the disk I/O completes), **look up** the needed information and process the event.
  - **Example:**
    - Store socked descriptor (`sd`) in a hash table indexed by file descriptor (`fd`)
    - When I/O completes, use `fd` to access `sd`
    - Send the data to the `sd`

What is still difficult with Events.

- Systems moved from a single CPU to multiple CPUs.
  - Some of the simplicity of the event-based approach disappeared.

- It does not integrate well with certain kinds of systems activity.
  - **Ex. Paging**: A server will not make progress until page fault completes (implicit blocking).

- Hard to manage overtime: The exact semantics of various routines changes.

- Asynchronous disk I/O never quite integrates with asynchronous network I/O in as simple and uniform a manner as you might think.

- Disclaimer: Disclaimer: This lecture slide set is used in AOS course at University of Cantabria by V.Puente. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)