Alternative Concurrency Models

COS 417: Operating Systems

Spring 2025, Princeton University

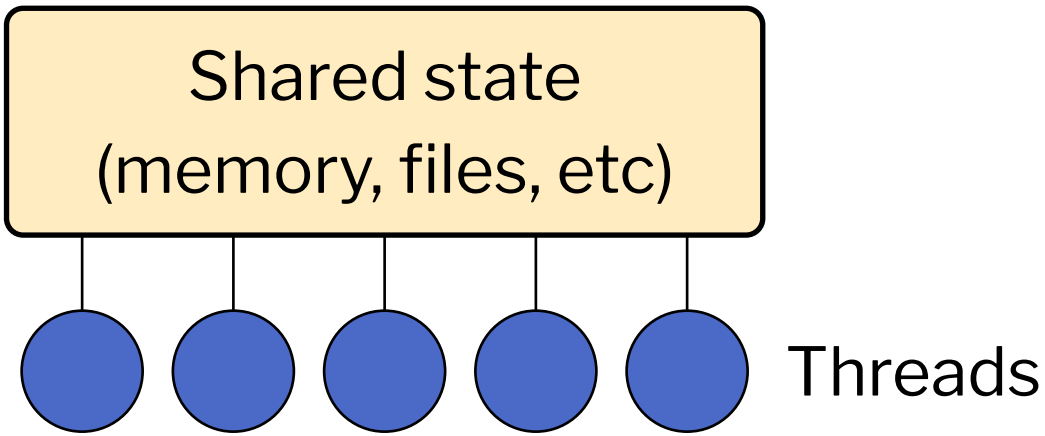
Why do we want concurrency?

- Anything you can do concurrently, you can do sequentially, just slower
- Overlap computations with slow I/O
- Responsiveness
  - Avoid starvation
- Harness parallelism in hardware
  - Multiple CPU cores, pipelining, etc

Why do we want concurrency?

- Anything you can do concurrently, you can do sequentially, just slower
- Overlap computations with slow I/O
- **Responsiveness**
  - Avoid starvation
- **Harness parallelism in hardware**
  - Multiple CPU cores, pipelining, etc

What are threads?



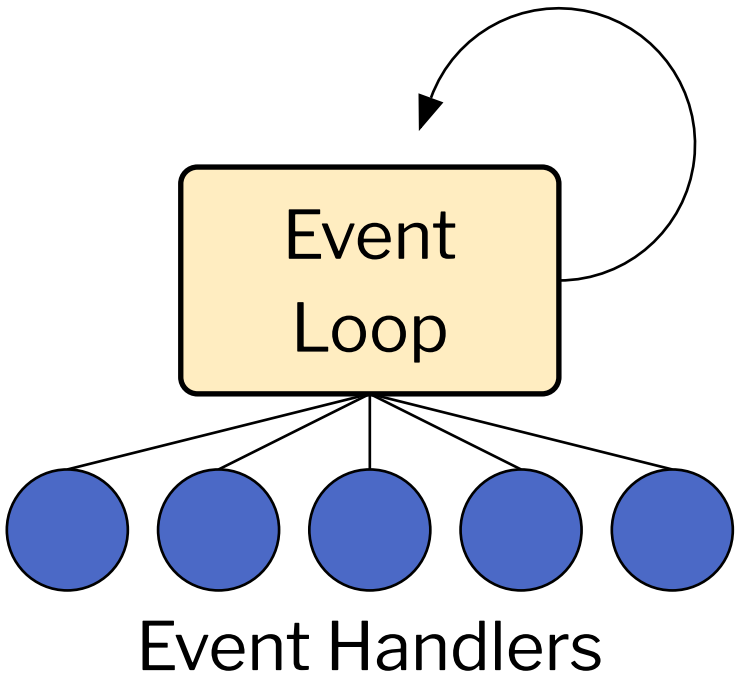
- General purpose solution for managing concurrency
- Multiple independent execution streams
- Shared state
- Pre-emptive scheduling
- Synchronization (e.g. locks, condition variables, semaphores…)

```c
/* Some details elided for brevity */
void *echo(int client_fd) {
    char buf[BUFFER_SIZE];
    int read_bytes = recv(client_fd, buf, BUFFER_SIZE, 0);
    send(client_fd, buff, read_bytes, 0);
}

int main() {
    int server_fd = socket(PF_INET, SOCK_STREAM, 0);
    bind(server_fd, &serveraddr, sizeof(serveraddr));
    listen(server_fd, 8080);
    while (1) {
        int client_fd = accept(server_fd);
        pthread_t child_tid;
        pthread_create(&child_tid, NULL, echo, client_fd);
    }
}
```

Events, an Alternative

- One execution stream
  - no CPU concurrency
- Register interest in events
- Event loop:
  1. waits for events
  2. invokes handlers
- No preemption
- Handlers generally short-lived



```
/* Many details elided for brevity */

void accept_cb(ev_loop *loop, ev_io *w, int revents);
void read_cb(ev_loop *loop, ev_io *w, int revents);

int main() {
    int server_fd = socket(PF_INET, SOCK_STREAM, 0);
    bind(server_fd, &serveraddr, sizeof(serveraddr));
    listen(server_fd, 8080);

    struct ev_loop *loop = ev_default_loop(0);
    ev_io w_accept;
    ev_io_init(&w_accept, accept_cb, sd, EV_READ);
    ev_io_start(loop, &w_accept);
    while(1) ev_loop(loop, 0);
}
```

```c
/* Many details elided for brevity */
void accept_cb(ev_loop *loop, ev_io *w, int revents) {
  int client_sd = accept(w->fd);
  ev_io *w_client = (struct ev_io*) malloc (sizeof(struct ev_io));
  ev_io_init(w_client, read_cb, client_sd, EV_READ);
  ev_io_start(loop, w_client);
}
void read_cb(ev_loop *loop, ev_io *w, int revents) {
  char buffer[BUFFER_SIZE];
  int read_bytes = recv(watcher->fd, buffer, BUFFER_SIZE, 0);
  send(client_fd, buff, read_bytes, 0);
}
```

Why Threads are a Bad Idea

(for most purposes)

Credit to John Ousterhout, Sun Microsystems 1995

Main argument

1. Threads are only better when CPU concurrency is truely needed.
2. True CPU concurrency is usually not needed.

What are threads used for?

- **Operating Systems**: one kernel therad for each user process.
- **Scientific applications**: one thread per CPU core (harness hardware parallelism)
- **Distributed Systems**: process requests concurrently (overlap I/O)
- **GUIs**:
  - Threads correspond to user actions
  - Service display during long-running computations

But threads are too hard...

Why Threads are Hard

- **Synchronization**:
  - Must coordinate access to shared data with locks
  - Forget a lock? Corrupted data…
- **Deadlock**:
  - Circular dependencies among locks
  - Each process is waiting on some other process: system hangs

Why Threads are Hard

- **Hard to debug**
  - data dependencies, timing dependencies
- **Threads break modularity**
  - can’t design modules independantly
- **Callbacks don’t work with locks**
- **Achieving good performance is hard**
  - Coarse grain locking yields low concurrency
  - Fine-grain locking increases complexity, reduces performance und
  - OSs limit performance (scheduling, context switches)

What are events used for?

- **GUIs**:
  - Threads correspond to user actions
  - Service display during long-running computations
- **Web Programming**:
- JavaScript is event based
- **I/O-Parallel Services**:
  - Web servers, network middleboxes, distributed storage, …
  - One handler for each source of input
  - Handler processes incoming request, sends response
  - Asynchronous I/O for I/O overlap

Problem With Events

- **Long running event handlers** make applications non-responsive
  - Break up handlers (e.g. event-driven I/O)
  - Create a thread for long-running computations, use events for completion
- **Loose implicit state tracking**
  - You need to *rip the stack*
    - **No CPU concurrency**
- Event-drive I/O not always well supported
  - E.g., asynchronous file I/O notoriously poor on Linux

Events vs. Threads

- **Concurrency**:
  - Events avoid it as much as possible
  - Sharing data with events is easy in simple cases
  - Sharing data with threads is hard in simple cases
- **Debugging**
  - Event timing entirely in event arrival
  - Thread timing related to scheduler decisions
- **Performance**
  - Events faster than threads on single CPU
  - Threads can take advantage of multiple CPU cores

Conclusion

Only use threads when you need true CPU concurrency

In Defense of Threads

Why Events are a Bad Idea (for high-concurrency servers), Behren, Condit & Brewer, HotOS 2003

Main argument

1. Performance issues with threads are implementation artifacts, or due to preemption.
2. Threads are actually easier to program

Thread's Performance Woes

Criticism: Switching between threads is slow

Analysis: Source is threefold:

- **Preemption**
- Kernel crossings
- Implementations not designed for high concurrency

Solution: Build a thread library

- in userspace
- with no preemption
- eliminate O(n) operations.

High overhead for Synchronization

Criticism: event synchronization is free

Analysis: it's co-operative scheduling!

- No synchronization needed with cooperative scheduling
- *And* only on uniprocessors

Solution: Already solved with co-operative threads

High-level of Abstraction Forces Suboptimal Scheduling

Criticism: Thread schedulers have to be generic.

Analysis: It's the kernel's fault!

- Thread packages re-use the kernel scheduler
- Kernel doesn't have application-level information

Solution: Can make optimal scheduling decisions in user-level threads package.

Threads could be *as* **efficient as events!***

*If they are non-preemptive and scheduled and in userspace

Benefits of Threads

- More natural control flow in most cases:
  - Call/return in events requires asynchrony
- Exception handling and cleanup
  - Need to clean up live state on exceptions
  - Thread stack makes this trivial
  - Event state somewhere on heap

What's the state of the art today?

Closures

```
function main() {
  let server = net.createServer();
  server.on('connection', (socket) => {
    socket.on('data', (data) => {
      socket.write(data);
    });
  })
}
```

Coroutines

```
func main() {
  listener, err := net.Listen("tcp", 8080)
  for {
    conn, err := listener.Accept()
    go handleConnection(conn)
  }
}

func handleConnection(conn net.Conn) {
  for {
    bytes, _ := reader.ReadBytes(byte('\n'))
    conn.Write([]byte(line))
  }
}
```