Event-Based Programming

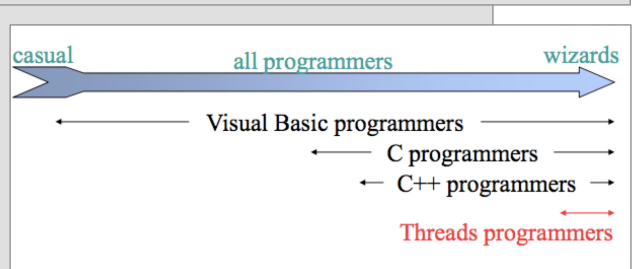
- Threads are “Hard”
- RECAP: Cooperative Task Management
- Event-driven Programming
- Event: Advantages
- Events: Problems

Threads have Limitations

John Ousterhout on Threads:

Threads are wrong to use:

- Threads are “Too hard for most programmers to use.”
- “Even for experts, development is painful.”



John Ousterhout: “Why Threads Are a Bad Idea (for most purposes)”, Sept. 1995

Threads have Limitations

John Ousterhout on Threads:

Threads are wrong to use:

- Threads are "Too hard for most programmers to use."
- "Even for experts, development is painful."

Conclusion:

- "Threads should be used only when true CPU concurrency (parallelism) is needed."
- "For most purposes proposed for threads, events are better"

John Ousterhout: "Why Threads Are a Bad Idea (for most purposes)", Sept. 1995

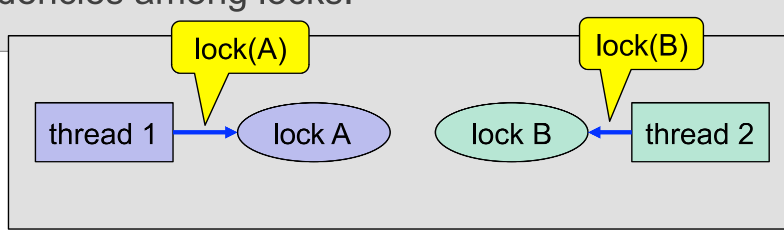
Why Threads are "Hard"

Synchronization:

- Must synchronize to ensure inter-thread invariants at all times.
- (*) Forget a lock? Invariant violated. (In English: data corrupted)

Deadlocks:

- (*) Circular dependencies among locks.



John Ousterhout: "Why Threads Are a Bad Idea (for most purposes)", Sept. 1995

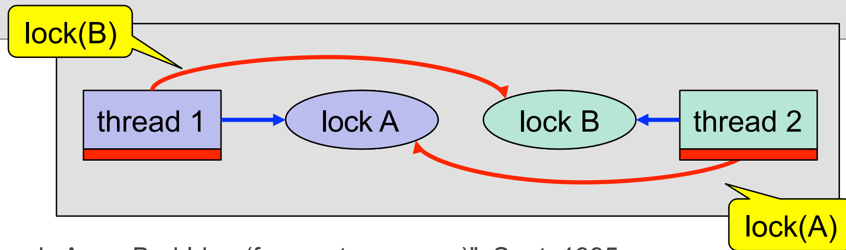
Why Threads are "Hard"

Synchronization:

- Must synchronize to ensure inter-thread invariants at all times.
- (*) Forget a lock? Invariant violated. (In English: data corrupted)

Deadlocks:

- (*) Circular dependencies among locks.

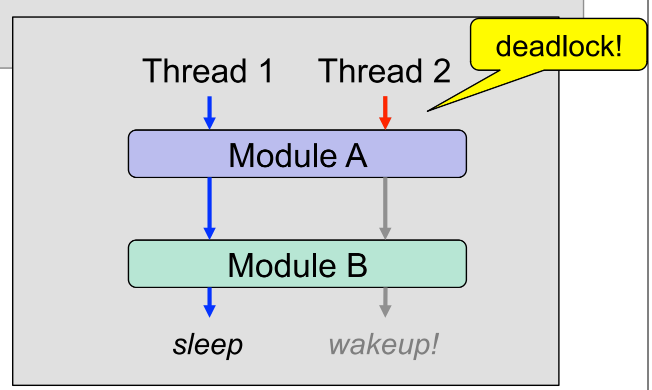


John Ousterhout: "Why Threads Are a Bad Idea (for most purposes)", Sept. 1995

Why Threads are "Hard"

"Lack of Composability":

- (*) Cannot design modules independently.

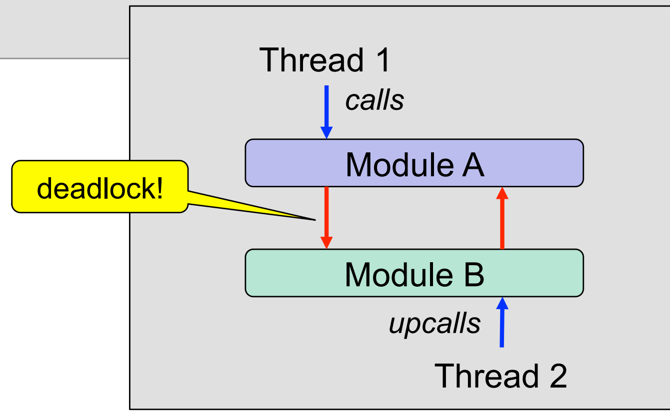


John Ousterhout: "Why Threads Are a Bad Idea (for most purposes)", Sept. 1995

Why Threads are "Hard"

"Locks don't work with Callbacks":

(*)



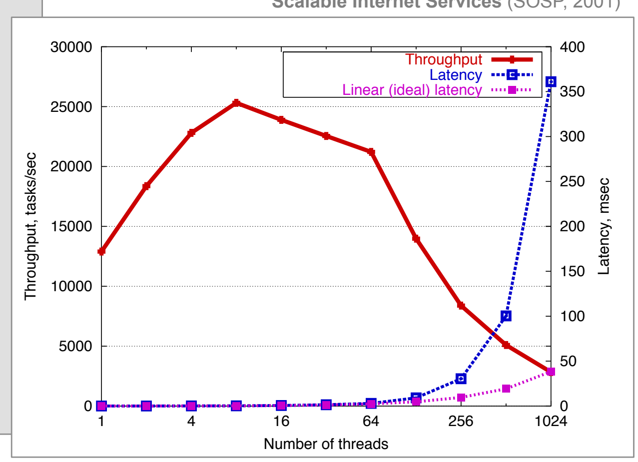
John Ousterhout: "Why Threads Are a Bad Idea (for most purposes)", Sept. 1995

Why Threads are "Hard"

"Achieving good performance is hard":

- (*) Simple locking (e.g. monitors) yields low concurrency.
- (*) Fine-grain locking increases complex and reduces performance in normal case.
- OS limits performance (scheduling, context switches, etc.)

Figure from: M. Welsh, D. Culler, and E. Brewer, **SEDA: An Architecture for Well Conditioned, Scalable Internet Services** (SOSP, 2001)



John Ousterhout: "Why Threads Are a Bad Idea (for most purposes)", Sept. 1995

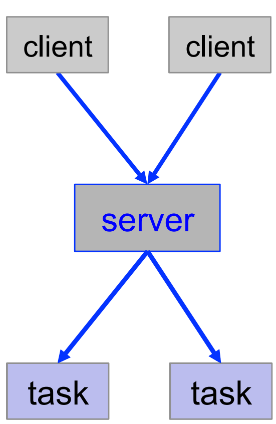
Why Threads are “Hard”

Threads “not well supported”:

- Hard to port threaded code.
- Standard libraries often not thread-safe.
- Kernel calls, window systems often not multi-threaded.
- Debugging tools are somewhat lacking.

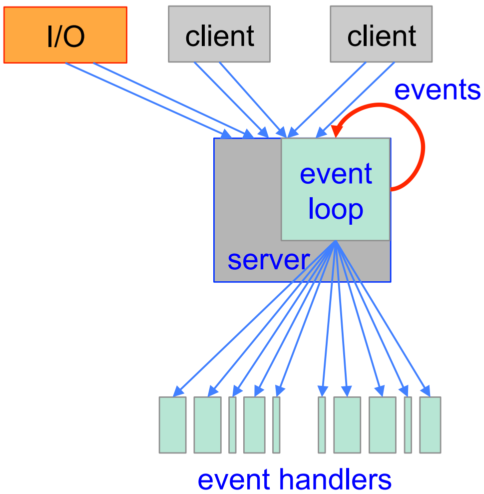
John Ousterhout: “Why Threads Are a Bad Idea (for most purposes)”, Sept. 1995

Cooperative Task Management



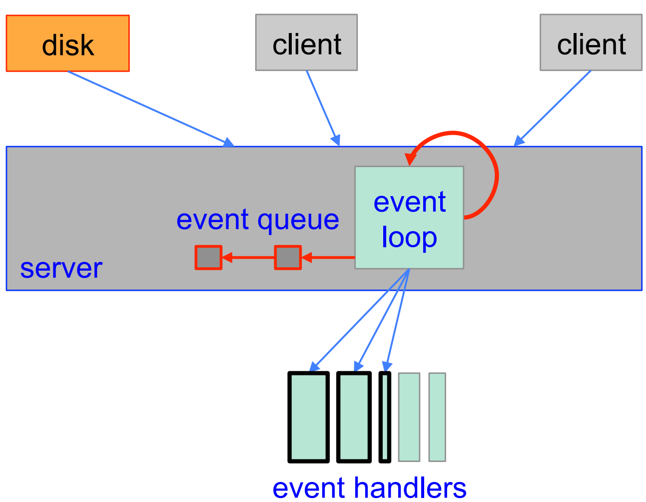
- **Serial** Task Management:
  - Execute each task to completion before starting new task.
- **Cooperative** Task Management:
  - (Voluntarily) yield CPU at well-defined points in execution.
- **Preemptive** Task Management:
  - Execution of tasks can interleave.

Event-Driven Programming

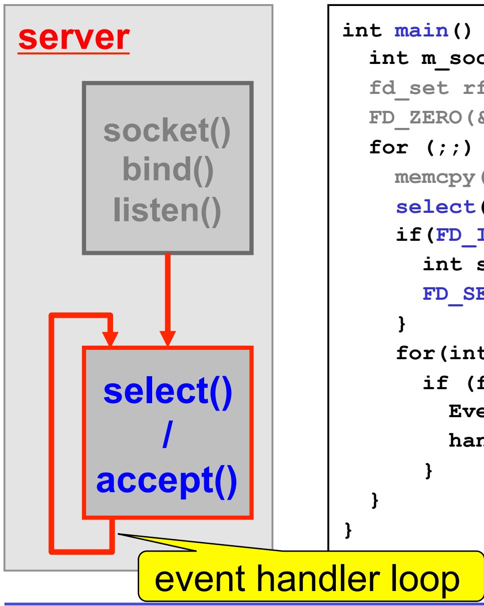


- Handlers "register" for events
- Event loop waits for events and then invokes handlers (through function calls)
- Handlers execute to completion (no preemption)
- Handlers are typically short-lived.
- Events can be initiated by devices as well

Event-Driven Programming: Step-by-Step



Example: Event Programming using select/accept




```
int main() {
  int m_sock = passiveTCPsock(...); /* master socket */
  fd_set rfds, afds;
  FD_ZERO(&afds);  FD_SET(m_sock, &afds);
  for (;;) { /* event loop */
    memcpy(&rfds, &afds, sizeof(rfds));
    select(nfds, &rfds, 0, 0, 0);
    if(FD_ISSET(m_sock, &rfds) { /* new connection */
      int s_sock = accept(m_sock, ...)); /* "slave" socket */
      FD_SET(s_sock, &afds);
    }
    for(int fd = 0; fd < nfds; fd++)
      if (fd != m_sock && FD_ISSET(fd, &rfds)) {
        EventHandler * handler = identify_handler(fd);
        handler.handle_event(fd);  /* invoke event handler */
      }
  }
}
```

Events: Advantages

Easier to use:

- No concurrency, no preemption, no synchronization, no deadlocks.

Easier to debug:

- Timing dependencies all generated by events, not by internal scheduling.
- Problems easier to trace, e.g. slow event handling vs. corrupted memory.

Fast:

- No context switching, no scheduling overhead.

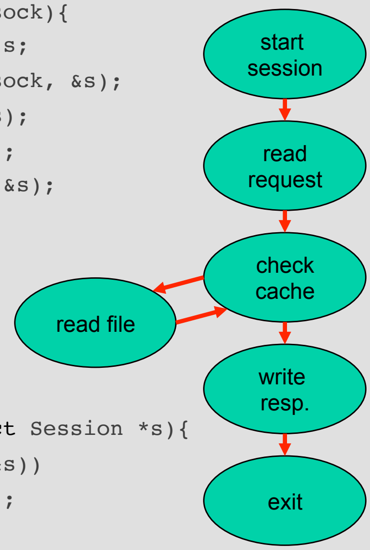
Portable:

- It's just function calls, after all.

John Ousterhout: "Why Threads Are a Bad Idea (for most purposes)", Sept. 1995

Events: Manual Control Flow Management

```
thread_main(int sock){
  struct Session s;
  start_session(sock, &s);
  read_request(&s);
  check_cache(&s);
  write_response(&s);
}
```

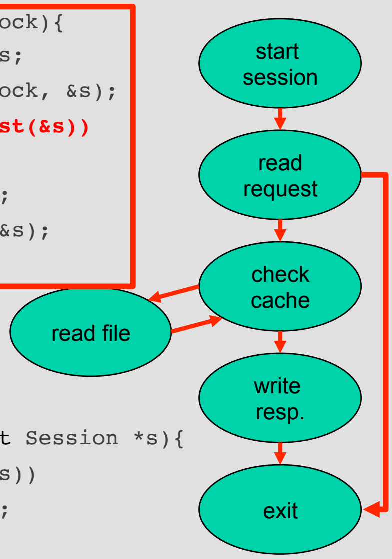



```
check_cache(struct Session *s){
  if (!in_cache(&s))
    read_file(&s);
}
```

```
StartHandler(event e){
  struct Session * s = new_session(e);
  RequestHandler.enqueue(s);
}
RequestHandler(struct Session *s){
  ...; CacheHandler.enqueue(s);
}
CacheHandler(struct Session *s){
  if (in_cache(&s))
    ResponseHandler.enqueue(s);
  else
    ReadFileHandler.enqueue(s);
}
ExitHandler(struct Session *s){
  ...; free_session(s);
}
```


Events: Manual Control Flow Management


```
thread_main(int sock){
  struct Session s;
  start_session(sock, &s);
  if (!read_request(&s))
    return;
  check_cache(&s);
  write_response(&s);
}
```

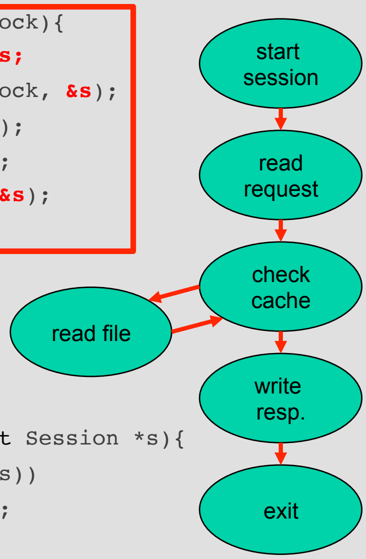



```
check_cache(struct Session *s){
  if (!in_cache(&s))
    read_file(&s);
}
```

```
StartHandler(event e){
  struct Session * s = new_session(e);
  RequestHandler.enqueue(s);
}
RequestHandler(struct Session *s){
  ...; if (error) ExitHandler(s);
  CacheHandler.enqueue(s);
}
CacheHandler(struct Session *s){
  if (!in_cache(&s))
    ReadFileHandler.enqueue(s);
  else
    ResponseHandler.enqueue(s);
}
ExitHandler(struct Session *s){
  ...; free_session(s);
}
```

Events: Manual Stack Management

```
thread_main(int sock){
  struct Session s;
  start_session(sock, &s);
  read_request(&s);
  check_cache(&s);
  write_response(&s);
}
```




```
check_cache(struct Session *s){
  if (!in_cache(&s))
    read_file(&s);
}
```

```
StartHandler(event e){
  struct Session * s = new_session(e);
  RequestHandler.enqueue(s);
}
RequestHandler(struct Session *s){
  ...; CacheHandler.enqueue(s);
}
CacheHandler(struct Session *s){
  if (!in_cache(&s))
    ReadFileHandler.enqueue(s);
  else
    ResponseHandler.enqueue(s);
}
ExitHandler(struct Session *s){
  ...; free_session(s);
}
```

Events: Problems – Summary

Manual control flow management:

- Control flow for single task is broken up across multiple procedures.

Manual stack management:

- Have to manually carry local state across these procedures.

Unexpected blocking:

- What if event handler is blocked by page fault?

No parallelism:

- How to run events on a multiprocessor?

Event-Based Programming

- Threads are “Hard”
- RECAP: Cooperative Task Management
- Event-driven Programming
- Event: Advantages
- Events: Problems