Concurrency

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)
Systems Software &
Architecture Lab.
Seoul National University

Spring 2019

Concurrency Mechanisms

- Processes
- Threads
- Events

Single-Threaded Echo Server

```
int main (int argc, char *argv[])
{
   . . .
   listenfd = socket(AF_INET, SOCK_STREAM, 0);
   bind(listenfd, (struct sockaddr *) &saddr, sizeof(saddr));
   listen(listenfd, 5);
   . . .

   while (1) {
      connfd = accept (listenfd, (struct sockaddr *)&caddr,
                       &caddrlen));
      while ((n = read(connfd, buf, MAXLINE)) > 0) {
         printf (“got %d bytes from client.\n”, n);
         write(connfd, buf, n);
      }
      close(connfd);
   }
}
```

Process-based Echo Server

```
int main (int argc, char *argv[])
{
   . . .
   signal (SIGCHLD, handler);

   while (1) {
      connfd = accept (listenfd, (struct sockaddr *)&caddr,
                       &caddrlen));
      if (fork() == 0) {
         close(listenfd);
         while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf (“got %d bytes from client.\n”, n);
            write(connfd, buf, n);
         }
         close(connfd);
         exit(0);
      }
      close(connfd);
   }
}
```

```
void handler(int sig) {
   pid_t pid;
   int stat;
   while ((pid = waitpid(-1, &stat,
                      WNOHANG)) > 0);
   return;
}
```

Thread-based Echo Server

```
int main (int argc, char *argv[])
{
   int *connfdp;
   pthread_t tid;
   . . .

   while (1) {
      connfdp = (int *)
                malloc(sizeof(int));
      *connfdp = accept (listenfd,
          (struct sockaddr *)&caddr,
          &caddrlen));

      pthread_create(&tid, NULL,
          thread_main, connfdp);
   }
}
```

```
void *thread_main(void *arg)
{
  int n;
  char buf[MAXLINE];

  int connfd = *((int *)arg);
  pthread_detach(pthread_self());
  free(arg);

  while((n = read(connfd, buf,
             MAXLINE)) > 0)
    write(connfd, buf, n);

  close(connfd);
  return NULL;
}
```

Event-based Echo Server (1)

```
typedef struct {
   int   maxfd;                     // largest descriptor in read_set
   int   nready;                    // number of ready desc. from select
   fd_set read_set;                 // set of all active descriptors
   fd_set ready_set;                // subset of desc. ready for reading
} pool;

int main (int argc, char *argv[])
{
   int listenfd, connfd, val;
   pool p;

   ...
   listenfd = ...                  // socket(), bind(), listen()

   // initialize pool
   p.maxfd = listenfd;
   FD_ZERO(&p.read_set);
   FD_SET(listenfd, &p.read_set);
```

Event-based Echo Server (2)

```
  while (1) {
    p.ready_set = p.read_set;
    p.nready = select(p.maxfd+1, &p.ready_set, NULL, NULL, NULL);

    if (FD_ISSET(listenfd, &p.ready_set)) {
      connfd = accept (listenfd, (struct sockaddr *)&caddr,
                            &caddrlen));
      FD_SET(connfd, &p.read_set);
      if (connfd > p.maxfd) p.maxfd = connfd;
      p.nready--;

    }
    check_clients (listenfd, &p);
  }

}
```

Event-based Echo Server (3)

```
void check_clients (int listenfd, pool *p) {
   int s, n;
   char buf[MAXLINE];
   for (s = 0; s < p->maxfd+1 && p->nready > 0; s++) {
      if (s == listenfd) continue;
      if (FD_ISSET(s, &p->read_set) && FD_ISSET(s, &p->ready_set)) {
         p->nready--;
         if ((n = read(s, buf, MAXLINE)) > 0)
            write(s, buf, n);
         if (n == 0) {                                // EOF
            close(s);
            FD_CLR(s, &p->read_set);
            if (s == p->maxfd) {
               p->maxfd--;
               while (!FD_ISSET(p->maxfd, &p->read_set)) p->maxfd--;
            }
         }
      }
   }
}
```

`select()`, `poll()`, and `epoll()`

- `int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)`
  - O(n) operations
  - fdsets (e.g. `readfds`, etc.) are destroyed on return, must be rebuilt for next call
  - More portable

- `int poll (struct pollfd *fds, nfds_t nfds, const struct timespec *tmo_p, const sigset_t *sigmask)`
  - More efficient for large-valued or sparse file descriptors
  - The same array can be used for next call

- `int epoll_wait (int epfd, struct epoll_event *events, int maxevents, int timeout)`
  - `epoll_create()`, `epoll_ctl()`, and wait for events using `epoll_wait()`
  - O(1) operations: `epoll_wait()` returns only the objects with ready file descriptors
  - Linux-specific

Why Threads Are A Bad Idea? (for most purposes)

(John Ousterhout, A talk @ USENIX Technical Conference, 1996)

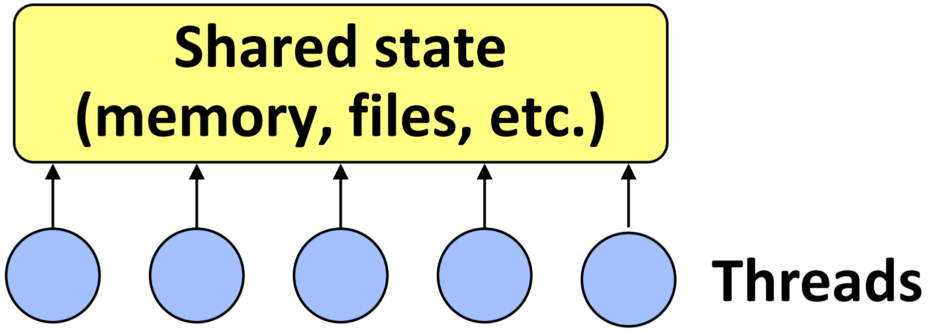
Some of slides are borrowed from the authors' presentation.

Introduction

- Threads
  - Grew up in OS world (processes)
  - Evolved into user-level tool
  - Proposed as solution for a variety of problems
  - Every programmer should be a threads programmer?
- Problem: threads are very hard to program
- Alternative: events
- Claims
  - For most purposes proposed for threads, events are better
  - Threads should be used only when true CPU concurrency is needed

What Are Threads?

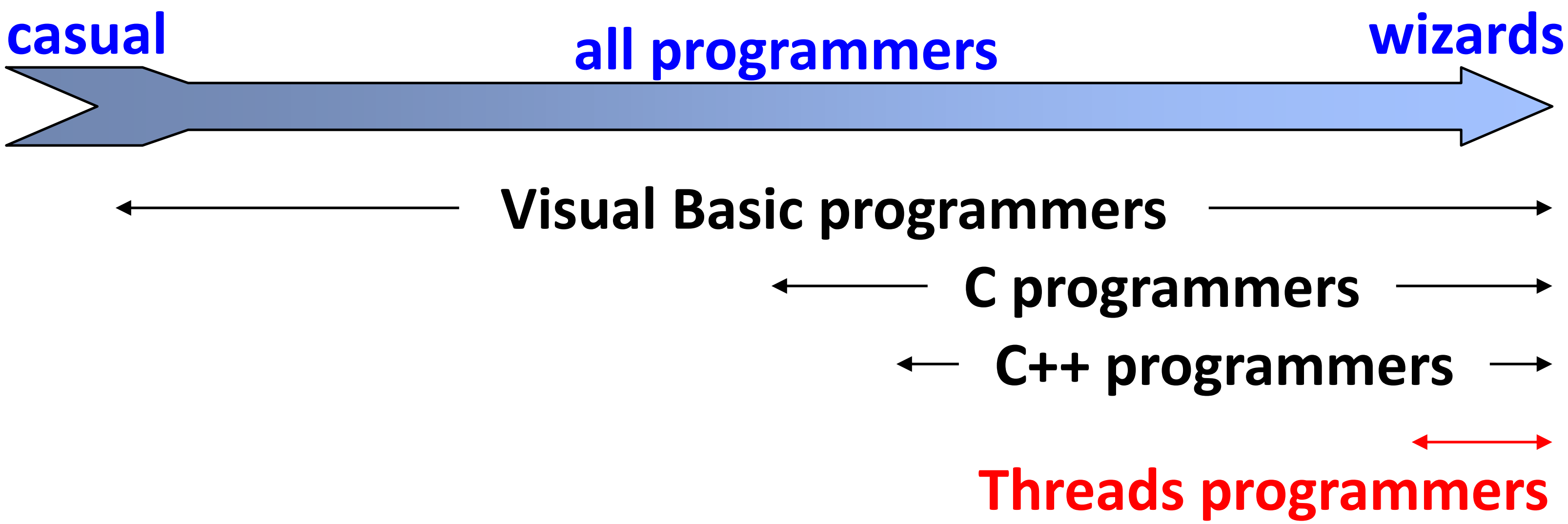
- General-purpose solution for managing concurrency
- Multiple independent execution streams
- Shared state
- Preemptive scheduling
- Synchronization (e.g. locks, conditions)



What Are Threads Used For?

- Operating systems:
  - One kernel thread for each user process
- Scientific applications:
  - One thread per CPU (solve problems more quickly)
- Distributed systems:
  - Process requests concurrently (overlap I/Os)
- GUIs
  - Threads correspond to user actions; can service display during long-running computations
  - Multimedia, animations, …

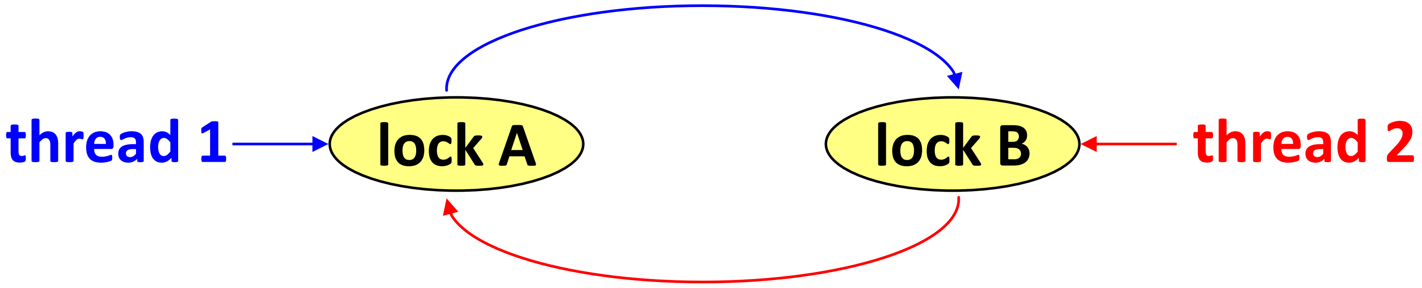
What's Wrong With Threads?



- Too hard for most programmers to use
- Even for experts, development is painful

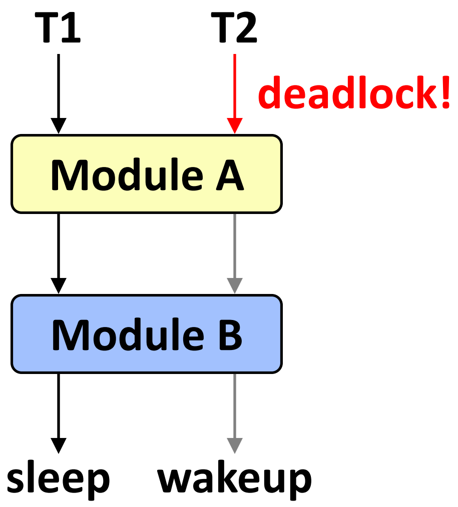
Why Threads Are Hard? (1)

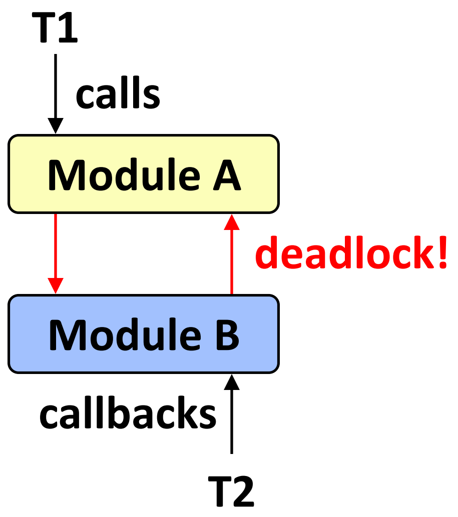
- Synchronization
  - Must coordinate access to shared data with locks
  - Forget a lock? Corrupted data
- Deadlock
  - Circular dependencies among locks
  - Each process waits for some other process: system hangs



Why Threads Are Hard? (2)

- Hard to debug
  - Data dependencies, timing dependencies
- Threads break abstractions
  - Can’t design modules independently
- Callbacks don’t work with locks



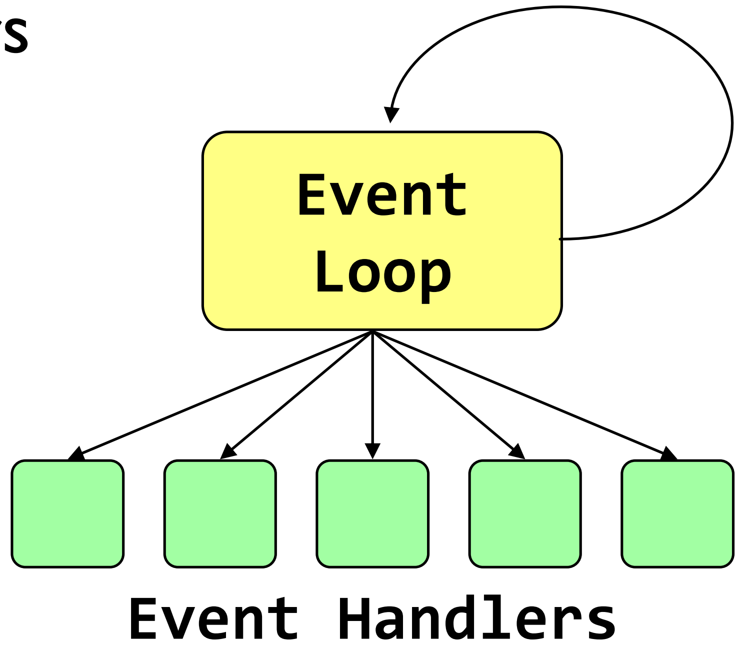


Why Threads Are Hard? (3)

- Achieve good performance is hard
  - Simple locking (e.g. monitors) yields low concurrency
  - Fine-grain locking increases complexity, reduces performance in normal case
  - OSes limit performance (scheduling, context switching)
- Threads not well supported
  - Hard to port threaded code (PCs? Macs?)
  - Standard libraries not thread-safe
  - Kernel-calls, window systems not multi-threaded
  - Few debugging tools (LockLint, debuggers?)
- Often don't't' want concurrency anyway
  - e.g. Window events

Event-driven Programming

- One execution stream: no CPU concurrency
- Register interest in events (callbacks)
- Event loop waits for events, invokes handlers
- No preemption of event handlers
- Handlers generally short-lived



What Are Events Used For?

- Mostly GUIs
  - One handler for each event (press button, invoke menu entry, etc.)
  - Handler implements behavior (undo, delete file, etc.)

- Distributed systems
  - One handler for each source of input (socket, etc.)
  - Handler processes incoming request, sends response
  - Event-driven I/O for I/O overlap

Problems with Events

- Long-running handlers make application non-responsive
  - Fork off subprocesses for long-running things (e.g. multimedia), use events to find out when done
  - Break up handlers (e.g. event-driven I/O)
  - Periodically call event loop in handler (reentrancy adds complexity)
- Can't maintain local state across events (handler must return)
- No CPU concurrency (not suitable for scientific apps)
- Event-driven I/O not always well supported (e.g. poor write buffering)

Events vs. Threads (1)

- Events avoid concurrency as much as possible, threads embrace
  - Easy to get started with events: no concurrency, no preemption, no synchronization, no deadlock
  - Use complicated techniques only for unusual cases
  - With threads, even the simplest application faces the full complexity

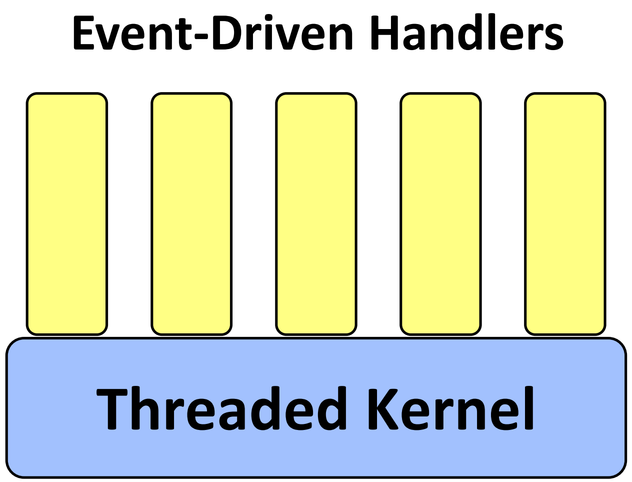
- Debugging easier with events
  - Timing dependencies only related to events, not to internal scheduling
  - Problems easier to track down: slow response to button vs. corrupted memory

Events vs. Threads (2)

- Events faster than threads on single CPU
  - No locking overheads
  - No context switching
- Events more portable than threads
- Threads provide true concurrency
  - Can have long-running stateful handlers without freezes
  - Scalable performance on multiple GPUs

Should You Abandon Threads?

- NO: important for high-end serves (e.g. databases)
- But, avoid threads whenever possible
  - Use events, not threads, for GUIs, distributed systems, low-end servers
  - Only use threads where true CPU concurrency is needed
  - Where threads needed, isolate usage in threaded application kernel: keep most of code single-threaded



Conclusion

- Concurrency is fundamentally hard; avoid whenever possible
- Threads more powerful than events, but power is rarely needed
- Threads much harder to program than events; for experts only
- Use events as primary development tool (both GUIs and distributed systems)
- Use threads only for performance-critical kernels

Why Events Are A Bad Idea?
(for high-concurrent servers)

(R. von Behren et al., HotOS, 2003)

Some of slides are borrowed from the authors' presentation.

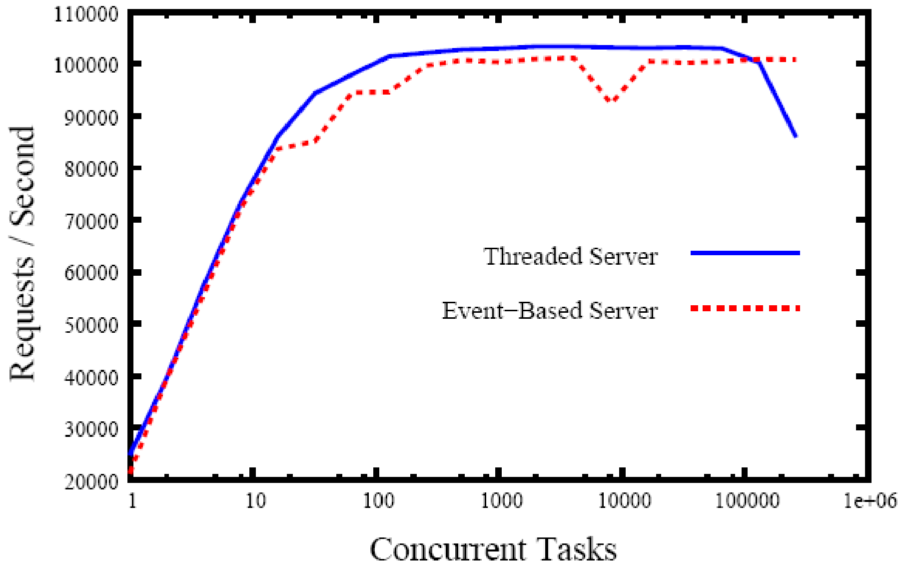
Introduction

- Four primary arguments for events
  - Inexpensive synchronization due to cooperative multitasking
  - Lower overhead for managing state (no stacks)
  - Better scheduling and locality, based on application-level information
  - More flexible control flow (not just call/return)

- Claim:
  - The right paradigm for highly concurrent applications is a thread package with better compiler support

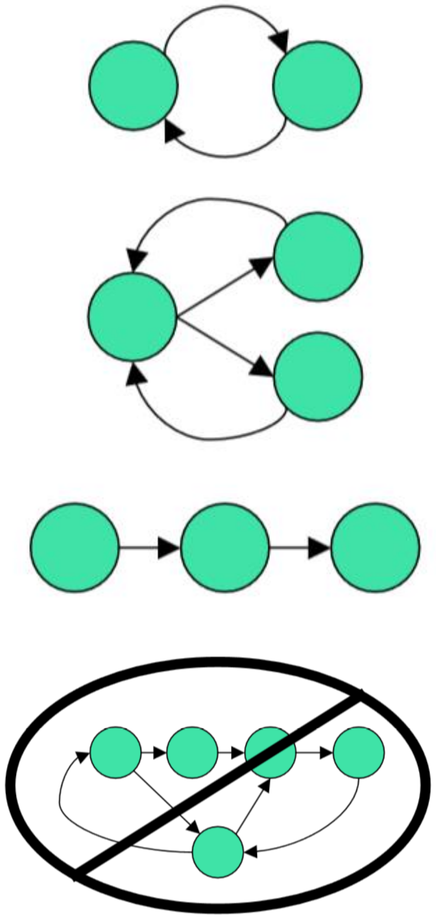
Threads: Performance

- Criticism: Many attempts to use threads for high concurrency have not performed well
- This is due to poor thread implementations
  - The presence of O(n) operations in scheduling, etc.
  - Relatively high context switch overhead (preemption, kernel crossings)
- They are not intrinsic properties of threads
  - SEDA threaded web server benchmark with modified GNU Pth user-level threads



Threads: Control Flow

- Criticism: Threads have restrictive control flow
- Complicated control flow patterns are rare in practice
  - Three simple categories: call/return, parallel calls, pipelines
  - All of these patterns can be expressed more naturally with threads
- Complex patterns
  - Hard to understand and error-prone
  - Lead subtle races
- Dynamic fan-in and fan-out
  - Multicast or publish/subscribe applications
  - Less graceful with threads
  - Not used in high-concurrency servers

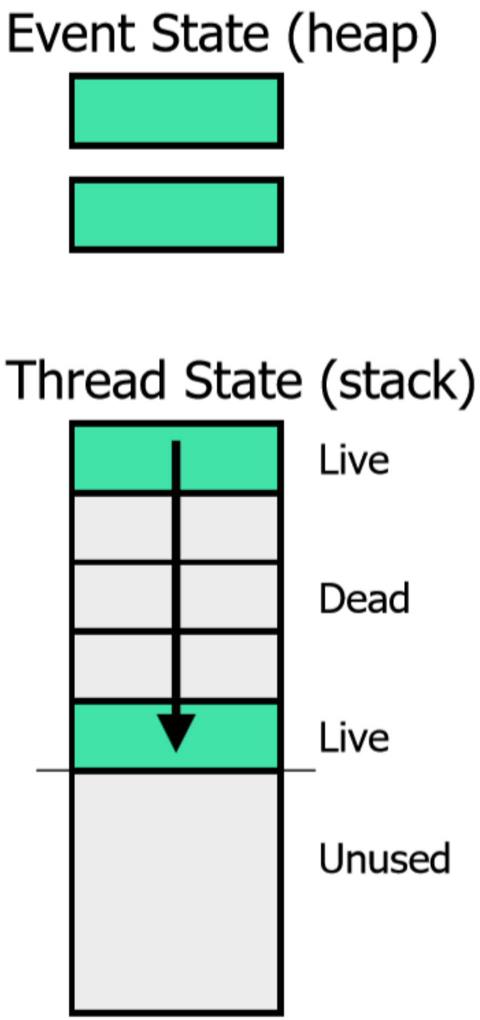


Threads: Synchronization

- Criticism: Thread synchronization mechanisms are too heavyweight
- Synchronization in event systems comes for free
  - Mainly due to cooperative multitasking, not events themselves
- Cooperative thread systems can have the same benefits
- In either cases, free only on uniprocessors

Threads: State Management

- Criticism: Thread stacks are an ineffective way to manage live state
- State management in threads
  - Stack overflow?
  - Wasting virtual address space on large stacks
  - Automatic state management via call stack allow programmers to be wasteful
- Can be solved:
  - Dynamic stack growth
  - Minimizing live state



Threads: Scheduling

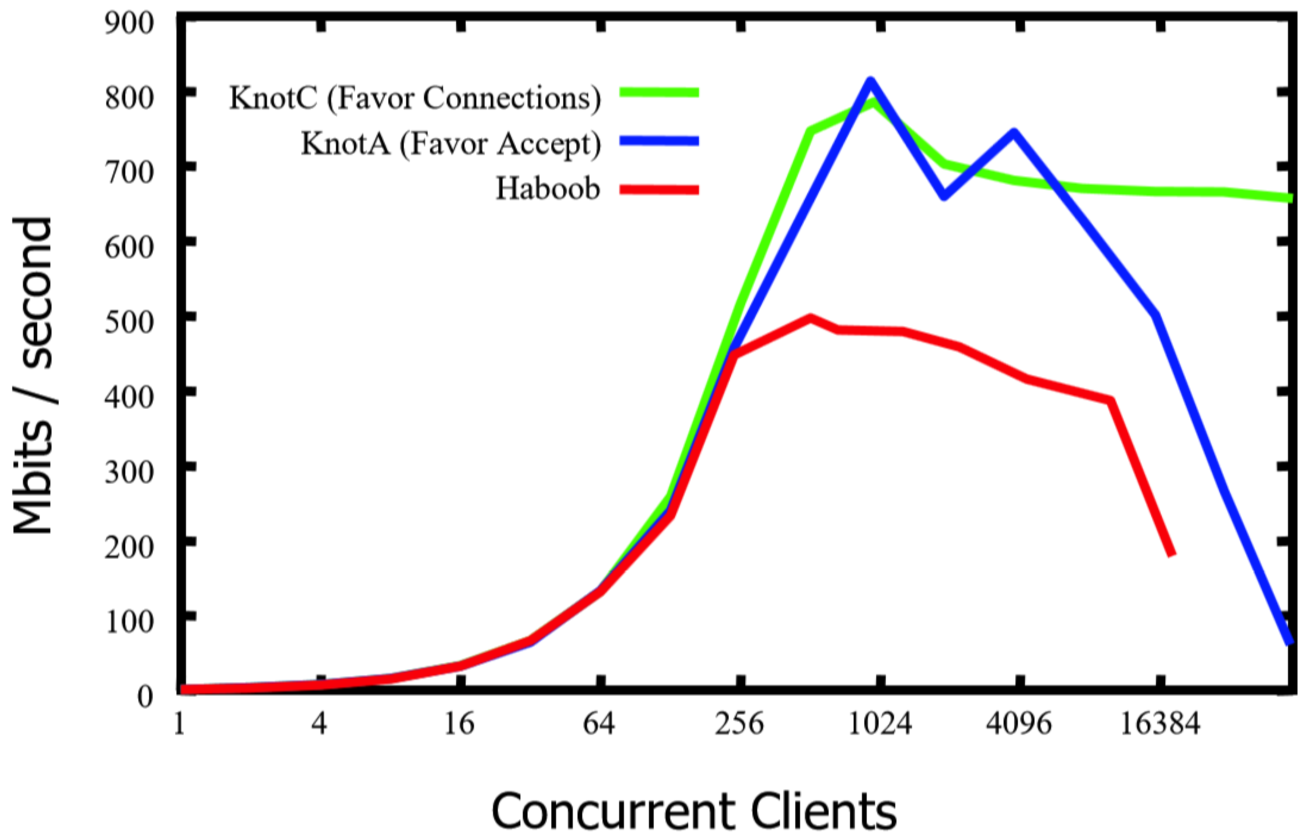
- Criticism: The virtual processor model provided by threads forces the runtime system to be too generic and prevents it from making optimal scheduling decisions
- Scheduling in event systems
  - Event systems are capable of scheduling event deliveries at application level
  - Events allow better code locality by running several of the same kind of event in a raw
- The same scheduling tricks can be applied to cooperative scheduled threads

Compiler Support for Threads

- Dynamic stack growth
  - Compiler analysis determines the amount of stack space needed
- Live state management
  - Compilers purge unnecessary state from the stack before making function calls
- Synchronization
  - Compilers can warn the programmer about data races

Evaluation

- User-level threads package
  - Subset of pthreads
  - Intercept blocking system calls
  - No O(n) operations
  - Support > 100K threads
  - 5000 lines of C code
- Simple web server: Knot
  - 700 lines of C code
- Similar performance
  - Linear increase, then steady
  - Drop-off due to `poll()` overhead



Conclusion

- Threads are actually a more appropriate abstraction for high-concurrency servers
  - The concurrency in modern servers results from concurrent requests that are largely independent
  - The code that handles each request usually sequential

- Small improvements to compilers and thread runtime system can eliminate the historical reasons to use events