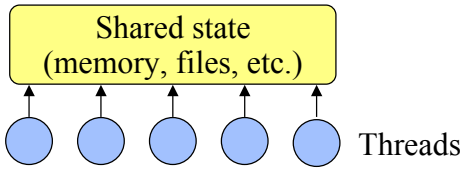
Events vs Threads

1. Lottery question: should I present the basic operation of lotteries? Coverd big picture on Tuesday.
2. Presentation: Sejal's group
3. Questions from reading
    a. What is pinning?
        i. Reference counting so resources can't be deleted
    b. How used?
    c. Tail-call optimization
    d. Closures:
        i. Set of things available locally. Normally gets lost when thread returns. Basic problem: to handle async events, want to get it back again when code returns
        ii. In Scheme: can explicitly create a function that is everything else in the function
4. Background: events and threads
    a. Threads:
        i. State: stack + registers
            1. Stack allocated without knowing how much space is needed, is conservative
            2. 

        ii. Scheduled by generic scheduler
            1. Kernel threads: preemptive (guarantees liveness, more or less)
            2. User threads: non-preemptive
        iii. Code style:
            1. Local state for computation stored on stack
            2. Invoke a sequence of (nested) functions to perform a computation
            3. Can block if doing blocking operation
        iv. Concurrency
            1. Need explicit concurrency mechanisms:
                a. Locks for mutual exclusion
                b. Condition variables / semaphores for coordination
    b. Example:
        i. `Do_request(request) {`
            1. Buffer = malloc();
            2. `If (!cached(request,buffer,&length) {`
                a. `Disk_read(request,buffer, &length)`

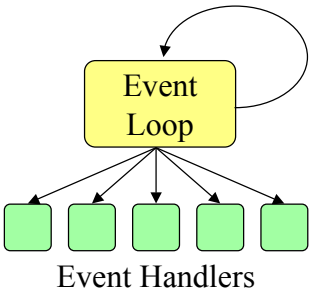
3. }
4. `send_response(request->socket,buffer,length);`
5. `close_socket(request->socket);`
6. `free(request->buffer);`

ii. Notes:

1. Thread may block on disk_read() and send_response()

c. Events:

i. How used:



- **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.**

ii.

1. Code runs non-preemptively (within the process) for compute-only
2. All blocking operations are asynchronous
3. Style 1: explicit events & handlers
    a. Code following blocking operation woken via generic event wakeup mechanism:
        i. handler = new event(callback_fn, param);
        ii. event = async_disk_read(request,buffer,&length);
        iii. register_handler(event,handler);
4. Style 2: callback functions as parameters, invoked on a separate stack later:
    a. event = async_disk_read(request,buffer,&length,callback_fn,param);
    b. Question: where does callback_fn run?
        i. When some thread decides to start running callbacks – e.g. SleepEx() in Windows
5. blocking function returns immediately. Async call signals handler when runs
6. "**short circuit"** – if code doesn't need to block, it can complete invoke the continuation immediately (synchronously) or return indicating it didn't block
7. One or more event loops detect signaling and invoke handler

8. "continuations" – the rest of the computation.
    a. For example, what you run when a function returns
    b. Or, if block, the remainder of the function

iii. Where used:
1. UI code: waiting for UI events
2. Network code: waiting for network packets

iv. Calling functions from event handler
1. Can call any non-blocking function
2. If call blocking function, need to split code and make everything after the blocking code a new event --- a "continuation"
    a. May duplicate code in original and continuation

v. State management:
1. Cannot store state on stack, as calling function returns to event loop when blocking
2. Need to save state in a structure passed as a parameter to event handler

vi. Concurrency:
1. Mutual exclusion:
    a. Can ensure that event handlers run to completion (no blocking operations) for uniprocessor
    b. If can partition handlers based on state, can run handlers without locks
    c. Event dispatcher guarantees only one handler at a time executes
2. Coordination
    a. Can explicitly signal event for following code, not need semaphores/mutex variables

vii. Note: implemented on top of kernel threads!

d. What are benefits of events?

i. Easier (?) synchronization/concurrency:
1. Not need locks
    a. Need to partition events to say which cannot be handled concurrently (assign a 'color')
2. Not need complex condition variable code
3. Fewer races, deadlocks

ii. Better control over scheduling
1. Can control which order events get handled in code
2. Example:
    a. In a web server, can decide to prioritize accepting socket connections over disk reads or prioritize some customers over others
3. Under overload:
    a. Can cancel events – signal error – rather than completing
    b. Gives control over how load is shed

c. Comparison: with threads (1 per request), just have to many thread, no way to cancel or control when scheduled

iii. Less memory consumption
1. Only allocate enough space for state needed in continuations rather than a worst-case stack

iv. More concurrency:
1. Blocking is explicit so never block with locks held

v. More modular
1. If make a function block, callers know because must pass in continuation (what to invoke when routine completes), but callers must change

vi. Can debug some kinds of problems easier
1. Latency: why isn't something running yet (what is in queue ahead)

e. What are benefits of threads?
i. Natural programming style for many people
ii. Easy state management: local variables
iii. More modular:
1. A function can block without changing code of calling function
2. But: not modular for performance, as making something block can hurt performance

iv. No need to decide what thread to run
1. Scheduling handled automatically by generic thread scheduler, runs everything eventually

v. Debugging
1. Have call stack indicating what thread has been doing
2. With events, hard to figure out why an event was not triggered, what it is waiting on.
a. "**Control flow is divided into many cooperatively-scheduled callback functions, obscuring context and programmer intent. This makes it hard to write event-driven programs and, worse, hard to analyze and debug them when they go wrong.**"
b. Hard to see what the path of execution was to an event, to understand what the order should be as code is not linear

5. Religious debate among people writing high-concurrency servers
a. Pro-event: John Ousterhout (Stanford, from Berkeley); Dave Mazieres, Frans Kaashoek, Robert Morris (MIT)
b. Pro-thread: Eric Brewer (Berkely)
i. Used threads for Inktomi search engine, sold for a billion dollars to Excite
ii. Now Chief of platforms at Google
c. Chief pro-event arguments:
i. Better concurrency control
ii. Better scheduling control
iii. Less memory usage

iv. Easier to program – less complex sync. Code
v. Faster: no preemptions, lock operations

d. Pro-thread arguments:
i. More modular (no stack ripping)
ii. Easier to debug
iii. Easier to program (more natural for some)
iv. Better exception handling; clear what to clean up

e. Resolution:
i. Make event code more like threads (this paper), MIT work
ii. Make thread code more like events (Capriccio from Berkeley) by
1. making stacks precise, make everything faster,
2. allow explicit thread scheduling based on resources from program (e.g., dispatch next event immediately rather than returning to generic event loop)
a. Switch from one thread to another specifically (switch_to()) rather than yielding to scheduler (yield())
b. Like LRPC handoff

6. This paper: how to combine event handling code with threaded code
a. Big goal: avoid "stack ripping" when calling blocking code
i. Allows modularity, reuse of existing code
b. Tool: cooperative threads
i. User-mode threads
ii. Explicit "switch-to" semantics; no general scheduler assumed
iii. Can be used to save state automatically rather than in continuations/events
c. Problem: stack ripping:
i. Given function:

```
CAInfo GetCAInfoBlocking(CAID caId) {
  CAInfo caInfo = LookupHashTable(caId);
  if (caInfo != NULL) {
    // Found node in the hash table
    return caInfo;
  }
  caInfo = new CAInfo();
  // DiskRead blocks waiting for
  // the disk I/O to complete.
  DiskRead(caId, caInfo);
  InsertHashTable(caId, CaInfo);
  return caInfo;
}
```

1.
2. Assumes automatic stack management: caInfo stored on stack along with caID to be invoked when disk read completes
ii. How make into a non-blocking code?
1. Split into two functions:

```
void GetCAInfoHandler1(CAID caId,
                Continuation *callerCont)
{
  // Return the result immediately if in cache
  CAInfo *caInfo = LookupHashTable(caId);
  if (caInfo != NULL) {
    // Call caller's continuation with result
    (*callerCont->function)(caInfo);
    return;
  }

  // Make buffer space for disk read
  caInfo = new CAInfo();
  // Save return address & live variables
  Continuation *cont = new
    Continuation(&GetCAInfoHandler2,
           caId, caInfo, callerCont);
  // Send request
  EventHandle eh =
       InitAsyncDiskRead(caId, caInfo);
  // Schedule event handler to run on reply
  // by registering continuation
  RegisterContinuation(eh, cont);
}
```

a.

b. Note it calls caller continuation when done; does not resume caller by just "returning"

c. State needed is passed into continuation

2. On disk-read complete:

```
void GetCAInfoHandler2(Continuation
*cont) {
  // Recover live variables
  CAID caId = (CAID) cont->arg1;
  CAInfo *caInfo = (CAInfo*) cont->arg2;
  Continuation *callerCont =
       (Continuation*) cont->arg3;
  // Stash CAInfo object in hash
  InsertHashTable(caId, caInfo);
  // Now "return" results to original caller
  (*callerCont->function)(callerCont);
}
```

a.

3. Note: need to take everything after the disk read, move to new event and make callable after disk read – "rip the stack"

d. Solution:

i. Big idea: lightweight threads to store state

1. Can easily switch to another thread or resume the thread

ii. Calling threaded code from event based code:

1. Run threaded code on another fiber

a. Allows returning immediately instead of blocking

2. Make threaded code signal following event when completes (like calling exit() when main() returns)
    a. Example: to invoke a call:

```
void VerifyCertCFA(CertData certData,
            Continuation *callerCont) {
  // Executed on MainFiber
  Continuation *vcaCont = new
           Continuation(VerifyCertCFA2,
                        callerCont);
  Fiber *verifyFiber = new
     VerifyCertFiber(certData, vcaCont);
  // On fiber verifyFiber, start executing
  // VerifyCertFiber::FiberStart
  SwitchToFiber(verifyFiber);
  // Control returns here when
  // verifyFiber blocks on I/O
}
```

    i.
    ii. verifyCertFiber invokes startfunction:

```
VerifyCertFiber::FiberStart() {
  // Executed on a fiber other than MainFiber
  // The following call could block on I/O.
  // Do the actual verification.
  this->vcaCont->returnValue =
        VerifyCert(this->certData);
  // The verification is complete.
  // Schedule VerifyCertCFA2
  scheduler->schedule(this->vcaCont);
  SwitchTo(MainFiber);
}
```

    iii.
        1. Note: verifyCert() invokes main fiber if it blocks, which resumes calling event (VeryCertCFA), so it doesn't block
    iv. Schedules continuation then returns to main fiber

```
void VerifyCertCFA2(Continuation
*vcaCont) {
  // Executed on MainFiber.
  // Scheduled after verifyFiber is done
  Continuation *callerCont =
         (Continuation*) vcaCont->arg1;
  callerCont->returnValue =
         vcaCont->returnValue;
  // "return" to original caller (FetchCert)
  (*callerCont->function)(callerCont);
}
```

    v.
    vi. Executes continuation from event-style code
    b. Net:
        i. thread blocking handled separately on a different thread with stack for local variables

ii. event code still uses explicit continuations, never blocks
1. good for concurrency rules (mutual exclusion, etc.)

e. Calling event code from threaded code
i. Big idea: suspend thread using fiber context switch; all state maintained on stack, resumed when event completes
ii. Example

```
Boolean GetCAInfoFCA(CAID caid) {
  // Executed on verifyFiber
  // Get a continuation that switches control
  // to this fiber when called on MainFiber
  FiberContinuation *cont = new
      FiberContinuation(FiberContinue,
                        this);
  GetCAInfo(caid, cont);
  if (!cont->shortCircuit) {
    // GetCAInfo did block.
    SwitchTo(MainFiber);
  }
  return cont->returnValue;
}
```

1.
2. If code completes without block, runs on same fiber and can return value directly
3. If it blocked, call returns, and continuation run from main fiber, need to switch back to blocking fiber. Need to switch to main fiber to let other things happen.

```
void FiberContinue(Continuation *cont) {
  if (!Fiber::OnMainFiber()) {
    // Manual stack mgmt code did not perform
    // I/O: just mark it as short-circuited
    FiberContinuation *fcont =
              (FiberContinuation) *cont;
    fcont->shortCircuit = true;
  } else {
    // Resumed after I/O: simply switch
    // control to the original fiber
    Fiber *f = (Fiber *) cont->arg1;
    f->Resume();
  }
}
```

4.
5. If on main fiber, switch back to calling fiber

f. Basically: run until event blocks, then switch to main fiber. Continuation will switch from main fiber back to thread
i. Allows thread-local variables to be preserved via explicit fibers

7. How use?
a. More-or-less automatic
b. What is needed?

i. Signature of function: how many parameters to store for invoking a fiber, what to do with return value
ii.