Example MoCs

**Sequential:**
- Finite state machines
- Pushdown automata
- Turing machines

**Functional:**
- Lambda calculus
- Recursive functions
- Combinatory logic
- Rewriting systems

**Concurrent:**
- Cellular automata
- Kahn process networks
- Petri nets
- Dataflow
- Actors
- CSP (rendezvous)

**Timed & Concurrent:**
- Synchronous/Reactive
- Discrete events
- Continuous time
Outline

• Definitions
• Threads
• Alternatives to Threads
• Deterministic Concurrency
Concurrent MoCs

“the rules that govern concurrent execution of the components and the communication between components”

http://ptolemy.org/systems
Concurrency

From the Latin, concurrere, “run together”
Concurrenty

**Google:**
- the fact of two or more events or circumstances happening or existing at the same time.

**Dictionary.com:**
- simultaneous occurrence; coincidence.

**Webster:**
- the simultaneous occurrence of events or circumstances
Points of Confusion

- The role of time
- Synchrony and asynchrony
- Concurrent vs. parallel
- Concurrent vs. nondeterministic
Layers of Abstraction for Concurrency in Programs

- Concurrent model of computation
- Dataflow, time triggered, synchronous, etc.
- Multitasking
  - Processes, threads, message passing
- Processor
  - Interrupts, pipelining, multicore, etc.
Uses of Concurrency in Software

- Reacting to external events (interrupts)
- Exception handling (software interrupts)
- Creating the illusion of simultaneously running different programs (multitasking)
- Exploiting parallelism in the hardware (multicore, VLIW, server farms)
- Dealing with real-time constraints (preemption, deadlines, priorities)
- Distributed computation (networked)
Outline

- Definitions
- Threads
- Alternatives to Threads
- Deterministic Concurrency
Threads are sequential concurrent procedures that share memory.

They have been the most commonly used mechanism for building concurrent software, but this is changing, for good reasons.

Processes are collections of threads with their own memory. Communication between processes occurs via OS facilities (like pipes, sockets, or files).
Without an OS, multithreading is achieved with interrupts. Timing is determined by external events.

Generic OSs (Linux, Windows, OSX, …) provide thread libraries (e.g. pthreads) and provide no guarantees about when threads will execute.

Real-time operating systems (RTOSs), like FreeRTOS, QNX, VxWorks, RTLinux, support a variety of ways of controlling when threads execute (priorities, preemption policies, deadlines, …).
Posix Threads (pthreads)

pthreads is an API (Application Program Interface) implemented by many operating systems, both real-time and not. It is a library of C procedures.

Standardized by the IEEE in 1988 to unify variants of Unix. Subsequently implemented in most other operating systems.

Some languages have threads built in, like Java, which uses pthreads under the hood.
Creating and Destroying Threads

#include <pthread.h>

void* threadFunction(void* arg) {
    ...
    return pointerToSomething or NULL;
}

int main(void) {
    pthread_t threadID;
    void* exitStatus;
    int value = something;
    pthread_create(&threadID, NULL, threadFunction, &value);
    ...
    pthread_join(threadID, &exitStatus);
    return 0;
}
What's Wrong with This?

#include <pthread.h>
#include <stdio.h>
void *my_thread() {
    int ret = 42;
    return &ret;
}

int main() {
    pthread_t task_id;
    void *status;
    pthread_create(&task_id, NULL, my_thread, NULL);
    pthread_join(task_id, &status);
    printf("%d\n", *(int*)status); return 0;
}
Threads can (and often do) share variables.

Threads may or may not begin running immediately after being created.

A thread may be suspended between any two *atomic* instructions (typically, assembly instructions, not C statements!) to execute another thread and/or interrupt service routine.

Threads can often be given *priorities*, but these may not be respected by the thread scheduler.

Threads may *block* on semaphores and mutexes.
A Scenario

Under Integrated Modular Avionics, software in the aircraft engine continually runs diagnostics and publishes diagnostic data on the local network.

Proper software engineering practice suggests using the observer pattern.

An observer process updates the cockpit display based on notifications from the engine diagnostics.
Typical thread programming problem

“The Observer pattern defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.”

*Design Patterns*, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides
(Addison-Wesley, 1995)
// Value that when updated triggers notification
// of registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element {
}
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {...}

// Procedure to update the value
void update(int newValue) {...}

// Procedure to call when notifying
void print(int newValue) {...}
// Value that when updated triggers notification of registered listeners.
int value;

// List of listeners. A linked list containing pointers to notify procedures.
typedef void* notifyProcedure(int);
typedef struct element {...
    notifyProcedure* listener;
    struct element* next;
} elementType;

elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {...}

// Procedure to update the value
void update(int newValue) {...}

// Procedure to call when notifying
void print(int newValue) {...}
// Value that when updated triggers notification of registered listeners.
int value;

// List of listeners. A linked list containing pointers to notify procedures.
typedef void* notifyProcedure(int);

struct element {
    // ...}

typedef struct element elementType;

elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {
    if (head == 0) {
        head = malloc(sizeof(elementType));
        head->listener = listener;
        head->next = 0;
        tail = head;
    } else {
        tail->next = malloc(sizeof(elementType));
        tail = tail->next;
        tail->listener = listener;
        tail->next = 0;
    }
}

// Procedure to update the value
void update(int newValue) {
    // ...}

// Procedure to call when notifying
void print(int newValue) {...}
// Value that when updated triggers notification of registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);

struct element {
    ...}

typedef struct element elementType;

elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {
    ...}

// Procedure to update the value
void update(int newValue) {
    value = newValue;
    // Notify listeners.
    elementType* element = head;
    while (element != 0) {
        (*(element->listener))(newValue);
        element = element->next;
    }
}

// Procedure to call when notifying
void print(int newValue) {
    ...}
Observer Pattern in C

// Value that when updated triggers notification of registered listeners.
int value;

// List of listeners. A linked list containing pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element {
...
} element;
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {
...
}

// Procedure to update the value
void update(int newValue) {
...
}

// Procedure to call when notifying
void print(int newValue) {
...
}

Will this work in a multithreaded context?
Will there be unexpected/undesirable behaviors?
Observer Pattern in C: How to make this thread safe?

```c
int value;

typedef void* notifyProcedure(int);
typedef struct element {…} elementType;
elementType* head = 0;
elementType* tail = 0;

void addListener(notifyProcedure listener) {
    if (head == 0) {
        head = malloc(sizeof(elementType));
        head->listener = listener;
        head->next = 0;
        tail = head;
    } else {
        tail->next = malloc(sizeof(elementType));
        tail = tail->next;
        tail->listener = listener;
        tail->next = 0;
    }
}

void update(int newValue) {…}

void print(int newValue) {…}
```
Using Posix mutexes on the observer pattern in C

However, this carries a significant deadlock risk. The update procedure holds the lock while it calls the notify procedures. If any of those stalls trying to acquire another lock, and the thread holding that lock tries to acquire this lock, deadlock results.
After years of use without problems, a Ptolemy Project code review found code that was not thread safe. It was fixed in this way. Three days later, a user in Germany reported a deadlock that had not shown up in the test suite.
#include <pthread.h>
...

pthread_mutex_t lock;

void addListener(notify listener) {
    pthread_mutex_lock(&lock);
    ...
    pthread_mutex_unlock(&lock);
}

void update(int newValue) {
    pthread_mutex_lock(&lock);
    value = newValue;
    ... copy the list of listeners ...
    pthread_mutex_unlock(&lock);
    elementType* element = headCopy;
    while (element != 0) {
        (*(element->listener))(newValue);
        element = element->next;
    }
}

int main(void) {
    pthread_mutex_init(&lock, NULL);
    ...
}
This is a very simple, commonly used design pattern. Perhaps Concurrency is Just Hard…

Sutter and Larus observe:

“Humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations.”

If concurrency were intrinsically hard, we would not function well in the physical world.

It is not concurrency that is hard...
Threads are sequential processes that share memory. From the perspective of any thread, the entire state of the universe can change between any two atomic actions (itself an ill-defined concept).

*Imagine if the physical world did that…*
What it Feels Like to Use Mutexes

Image "borrowed" from an Iomega advertisement for Y2K software and disk drives, Scientific American, September 1999.
Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans.

→ Need better ways to program concurrent systems

→ Better tools to analyze and reason about concurrency (e.g. model checking)
If the foundation is bad, then we either tolerate brittle designs that are difficult to make work, or we have to rebuild from the foundations.

Note that this whole thing is held up by threads.
Problems with the Foundations

A model of computation:

• Bits: $B = \{0, 1\}$
• Set of finite sequences of bits: $B^*$
• Computation: $f : B^* \rightarrow B^*$
• Composition of computations: $f \cdot f'$
• Programs specify compositions of computations

Threads augment this model to admit concurrency.

But this model does not admit concurrency gracefully.
Basic Sequential Computation

Formally, composition of computations is function composition.

Initial state: \( b_0 \in B^* \)

\[ b_n = f_n(b_{n-1}) \]

Final state: \( b_N \)
When There are Threads, Everything Changes

A program no longer computes a function.

\[ b_n = f_n( b_{n-1} ) \]

\[ b'_n = f_n( b'_{n-1} ) \]

Another thread can change the state.

Apparently, programmers find this model appealing because nothing has changed in the syntax.
Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer’s job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes) and limiting shared data accesses (e.g., OO design).
Incremental Improvements to Threads

- Object Oriented programming
- Coding rules (Acquire locks in the same order…)
- Libraries (Stapl, Java concurrent collections, …)
- Message passing (Actors, …)
- Publish and subscribe (ROS, MQTT, DDS, …)
- Transactions (Databases, …)
- Patterns (MapReduce, …)
- Formal verification (Model checking, …)
- Enhanced languages (Split-C, Cilk, Guava, …)
- Enhanced mechanisms (Promises, futures, asynchronous atomic callbacks …)
Threads are slowly getting replaced. E.g.:

- Asynchronous atomic callbacks
  - Python, Node.js, Vert.x, ...
- Actors
  - Akka, Orleans, Ray, ...
- Pub-Sub
  - ROS, Vert.x, DDS, ...
- ...

For concurrent programming to become mainstream, we must discard threads as a programming model. Nondeterminism should be judiciously and carefully introduced where needed, and it should be explicit in programs.

Lee, Berkeley
Message-passing programs may be better

```c
void* producer(void* arg) {
    int i;
    for (i = 0; i < 10; i++) {
        send(i);
    }
    return NULL;
}

void* consumer(void* arg) {
    while(1) {
        printf("received %d\n", get());
    }
    return NULL;
}

int main(void) {
    pthread_t threadID1, threadID2;
    void* exitStatus;
    pthread_create(&threadID1, NULL, producer, NULL);
    pthread_create(&threadID2, NULL, consumer, NULL);
    pthread_join(threadID1, &exitStatus);
    pthread_join(threadID2, &exitStatus);
    return 0;
}
```

But there is still risk of deadlock and unexpected nondeterminism!
A software component on a microprocessor in an aircraft door provides two network services:
1. “open”
2. “disarm”
Assume state is closed and armed.
What should it do when it receives a request “open”?
Outline

• Definitions
• Threads
• Alternatives to Threads
• Deterministic Concurrency
Asynchronous Atomic Callbacks

- Main event loop.
- Event handlers (“callbacks”) run to completion atomically.

Augment with worker threads that communicate with:

- Immutable data
- Publish-and-subscribe busses
Asynchronous Atomic Callbacks: Periodic Actions

- Shared variable x
- Timed actions on x
  - +1 every second
  - −2 every two seconds
- Observe every 4 seconds

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

On Node.js v5.3.0, MacOS Sierra:

0, 0, 0, 0, 0, −1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
NASA's Toyota Study Released by Dept. of Transportation released in 2011 found that Toyota software was “untestable.”

Possible victim of unintended acceleration.
Publish and Subscribe (Pub Sub)

Publisher

Broker

Subscriber

Publish to Topic: Temperature
Data: 18

Subscribe to Topic: Temperature

Data: 18

Subscriber

Subscriber

Subscriber

Etc.
Recall Challenge Problem

A software component on a microprocessor in an aircraft door provides two network services:

1. “open”
2. “disarm”

Assume state is closed and armed.

What should it do when it receives a request “open”?

Image from The Telegraph, Sept. 9, 2015
Another Answer to Threads: Actors

Actors are concurrent objects that communicate by sending each other messages.

- **Erlang** [Armstrong, et al. 1996]
- **Rebeca** [Sirjani and Jaghoori, 2011]
- **Akka** [Roestenburg, et al. 2017]
- **Ray** [Moritz, et al. 2018]
- ...
Outline

• Definitions
• Threads
• Alternatives to Threads
• Deterministic Concurrency
Ptolemy II has implementations of all of these and a few more with extensive demos.
Deterministic Concurrent MoCs

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)
Dataflow

- Computation Graphs [Karp, 1966]
- Dataflow [Dennis, 1974]
- Dynamic dataflow [Arvind, 1981]
- Structured dataflow [Matwin & Pietrzykowski 1985]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow and LabVIEW [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Dataflow synchronous languages [Lustre, Signal, 1980’s]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- ...

Jack Dennis
An actor with no inputs can fire at any time.
Dataflow Solution for Scheduling: Firing Rules

[Lee & Matsikoudis, 2009]

An actor with inputs has to specify at all times how many tokens it needs on each input in order to fire.
An actor inputs has to specify at all times how many tokens it needs on each input in order to fire.

When it fires, each reaction is invoked in a deterministic order.

[Lee & Matsikoudis, 2009]
Synchronous Dataflow Scheduling

When the firing rules and production patterns are static integer constants, then a lot of analysis and optimization is possible.

[Lee & Messerschmitt, 1986]
If execution times are also known, then throughput and latency bounds are derivable and optimal scheduling is possible (albeit intractable).

[Lee & Messerschmitt, 1986]
What should be the firing rule for Foo?

```
reactor Baz {
    input in;
    output out;
    reaction(in) {
        if (something) {
            send(out);
        }
    }
}
```
Buck [1993] showed that scheduling problems in general are undecidable in this framework.

Associate a symbolic variable with production and consumption parameters. Solve the scheduling problem symbolically. [Buck and Lee, 1993]
Various Dataflow Variants that Remain Decidable

- Cyclostatic dataflow [Lauwereins 1994]
- Parameterized dataflow [Bhattacharya & Bhattacharyya, 2001]
- Structured dataflow [Thies, 2002]
- Scenario-aware dataflow [Theelen, Geilen, Basten, et al. 2006]
- Reconfigurable dataflow [Fradet, Girault, et al., 2019]
A state machine governs the switching between production/consumption patterns and also execution times.

[Theelen, Geilen, Basten, et al. 2006]
Some Strategies

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)
A Different Solution: Blocking Reads

In Kahn Process Networks (KPN), every actor is a process that blocks on reading inputs until data is available.

```java
KPNActor Foo {
    input double, increment;
    int state = 1;
    while (true) {
        double
        read(double);
        state *= 2;
        increment
        x = read(increment);
        state += x;
        print state;
    }
}
```

[Kahn, 1974] [Kahn and MacQueen, 1977]
Blocking reads have trouble with data-dependent flow patterns

```kotlin
KPNActor Baz {
    input in;
    output out;
    while (true) {
        read(in);
        if (something) {
            send(out);
        }
    }
}

KPNActor Foo {
    input double, increment;
    int state = 1;
    while (true) {
        read(double);
        state *= 2;
        x = read(increment);
        state += x;
        print state;
    }
}
```
Blocking reads have trouble with data-dependent flow patterns

```cpp
KPNActor Baz {
    input in;
    output out;
    while (true) {
        read (in);
        if (something) {
            send (out);
        }
    }
}

KPNActor Foo {
    input double, increment;
    int state = 1;
    while (true) {
        if (something) {
            read (double);
            state *= 2;
        }
        x = read (increment);
        state += x;
        print state;
    }
}
```
Solution: Coordinated Control

Actor Baz {
    input in;
    output out;
    handler in() {
        if (something) {
            out.send();
        }
    }
}

Actor Foo {
    input double, increment;
    int state = 1;
    while (true) {
        if (something) {
            read(double);
            state *= 2;
        }
        x = read(increment);
        state += x;
        print state;
    }
}
Some Strategies

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)
An Alternative Approach to Coordination

Make the notion of the “absence” of a message as meaningful as its presence.
A Different Approach: Synchronous Languages

In the synchronous/reactive approach, there is a conceptual global “clock,” and on each “tick” of this clock, a connection either has a well-defined value or is “absent.” Each actor realizes a time-varying function mapping inputs to outputs.

[Benveniste & Berry, 1991]
At each tick of the clock, the job of the execution engine is to find a valuation $s$ for all signals such that $F(s) = s$.

This is called a fixed point of the function $F$. A theory of partial orders guarantees existence and uniqueness.

[Edwards and Lee, 2003]
Physically asynchronous, logically synchronous (PALS)

[Sha et al., 2009]
Some Strategies

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)
Discrete-Event Languages

DE is a generalization of SR, where there is a notion of “time between ticks.”

WARNING: immediately have (at least) two time lines: logical time and physical time(s).

[Lee & Zheng, 2007]
Discrete Events (DE)

- Events that are processed in timestamp order.
- Widely used in simulation
- Foundation of hardware description languages.
- A deterministic concurrent MoC.
- But how to realize on distributed machines?

And Lingua Franca!
References

Many dataflow papers: [https://ptolemy.berkeley.edu/publications/dataflow.htm](https://ptolemy.berkeley.edu/publications/dataflow.htm)

References