Pause 'n' play: asynchronous C♯ explained

Claudio Russo

Microsoft Research

Joint work with G. Bierman, G. Mainland (MSRC)
M. Torgersen and E. Meijer (Microsoft Corp.).

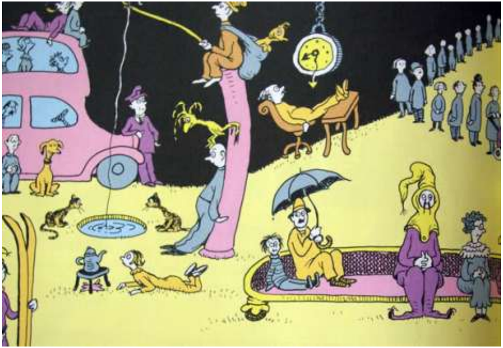
Outline

- Background and Motivation
- Example
- Semantics
- Concurrency
- Extensibility

Outline

- Background and Motivation
- Example
- Semantics
- Concurrency
- Extensibility

Waiting is no fun!



The Waiting Place…
…for people just waiting.
Waiting for a train to go
or a bus to come, or a plane to go
or the mail to come, or the rain to go
or the phone to ring, or the snow to snow
or waiting around for a Yes or a No
or waiting for their hair to grow.
Everyone is just waiting.
Waiting for the fish to bite
or waiting for wind to fly a kite
or waiting around for Friday night
or waiting, perhaps, for their Uncle Jake
or a pot to boil, or a Better Break
or a string of pearls, or a pair of pants
or a wig with curls, or Another Chance.
Everyone is just waiting.
NO!
That's not for you!
Dr Seuss, "Oh The Places You'll Go"

Visual Studio

C♯ 5.0

What problems do developers face today?

- Want "fluid" apps
- i.e. no hourglass/animated circle cursor!

Heart of the problem:

What problems do developers face today?

- Want “fluid” apps
- i.e. no hourglass/animated circle cursor!

Heart of the problem: **Synchronous operations**

- Simple to use but:
  - prevent progress until done,
  - reveal latency,
  - waste resources (calling threads).

What did my professor teach me?

Use **asynchronous** operations instead:

- enable concurrent progress whilst operations are running,
- hide latency,
- free up resources (calling threads).

What did my professor teach me?

Use **asynchronous** operations instead:

- enable concurrent progress whilst operations are running,
- hide latency,
- free up resources (calling threads).

Asynchronous coding has always been possible in C♯...

What did my professor teach me?

Use **asynchronous** operations instead:

- enable concurrent progress whilst operations are running,
- hide latency,
- free up resources (calling threads).

Asynchronous coding has always been possible in C♯...
...but it's never been easy.

What did my professor teach me?

Use **asynchronous** operations instead:

- enable concurrent progress whilst operations are running,
- hide latency,
- free up resources (calling threads).

Asynchronous coding has always been possible in C♯...
...but it's never been easy.

Today's message:

C♯ 5.0 makes asynchronous programming easy!

Asynchronous Operations

Synchronous operations block the caller until they're done.

An asynchronous operation separates:

- initiating the operation (without blocking).
- from waiting for its completion.

Asynchrony enables concurrency:

- do something else before waiting.
- initiate two things; wait in parallel for both.
- don't wait at all, but delegate your work!

Outline

Background and Motivation

Example

Semantics

Concurrency

Extensibility

Example: Reading from a stream

Synchronous:

```
int bytesRead = str.Read(...); // read and wait
```

Asynchronous: Task model (2008-)

```
Task<int> task = str.ReadAsync(...); // non-blocking
// do some work
int bytesRead = task.Result; // may block
```

(here, we use `task` as a blocking future.)

Efficient waiting

Efficient waiting uses callbacks, only invoked once done!

```
Task<int> task = str.ReadAsync(...); // non-blocking

task.ContinueWith(doneTask => { // task done!
   int bytesRead = doneTask.Result; // can't block
   ...
});
```

(here, we use **`task`** as a non-blocking promise.)

Efficient waiting

Efficient waiting uses callbacks, only invoked once done!

```
Task<int> task = str.ReadAsync(...); // non-blocking

task.ContinueWith(doneTask => { // task done!
   int bytesRead = doneTask.Result; // can't block
   ...
});
```

(here, we use `task` as a non-blocking promise.)

That’s the easy bit — the tough part is writing the callback.

Sync vs. async taste challenge

C♯ 4.0: Sync vs. Async

Synchronous stream length method

```
public static long Length(Stream src) {
  var buf = new byte[0x1000]; int bytesRead;
  long totalRead = 0;
  while ((bytesRead = src.Read(buf, 0, buf.Length)) > 0)
      totalRead += bytesRead;
  return totalRead;
}
```

C♯ 4.0: Sync vs. Async

Asynchronous stream length method

```
public static Task<long> LengthAsync(Stream src) {
  var tcs = new TaskCompletionSource<long>();
  var buf = new byte[0x1000]; int bytesRead;
  long totalRead = 0;
  Action<Task<int>> While = null;
  While = rdtask => {
    if ((bytesRead = rdtask.Result) > 0) {
        totalRead += bytesRead;
        src.ReadAsync(buf, 0, buf.Length).ContinueWith(While);
    }
    else tcs.SetResult(totalRead);
  };
  src.ReadAsync(buf, 0, buf.Length).ContinueWith(While);
  return tcs.Task;
}
```

C♯ 4.0: Sync vs. Async

Asynchronous stream length method

```
public static Task<long> LengthAsync(Stream src) {
  var tcs = new TaskCompletionSource<long>();
  var buf = new byte[0x1000]; int bytesRead;
  long totalRead = 0;
  Action<Task<int>> While = null;
  While = rdtask => {
    if ((bytesRead = rdtask.Result) > 0) {
        totalRead += bytesRead;
        src.ReadAsync(buf, 0, buf.Length).ContinueWith(While);
    }
    else tcs.SetResult(totalRead);
  };
  src.ReadAsync(buf, 0, buf.Length).ContinueWith(While);
  return tcs.Task;
}
```

"9/10 devs prefer the taste of synchronous code."

The problem with callbacks

Turning a synchronous call into an asynchronous call is hard! (The "Inversion of control" problem.)

Need to capture the next state of the caller as a callback.

Method state includes locals but also implicit control state:

- program counter (what to do next in this method)
- runtime stack (caller to return to from this method!)

This is easier in languages with call/cc (Scheme/SMLNJ).

Others have to be clever.

(Honorable mentions: Haskell, F#, Scala)

Sync vs. async taste challenge, again

C♯ 5.0: Sync vs. Async

Synchronous stream length method

```
public static long Length(Stream src) {
  var buf = new byte[0x1000]; int bytesRead;
  long totalRead = 0;
  while ((bytesRead = src.Read(buf,0,buf.Length))>0)
      totalRead += bytesRead;
  return totalRead;}
```

C♯ 5.0: Sync vs. Async

Synchronous stream length method

```
public static long Length(Stream src) {
  var buf = new byte[0x1000]; int bytesRead;
  long totalRead = 0;
  while ((bytesRead = src.Read(buf,0,buf.Length))>0)
      totalRead += bytesRead;
  return totalRead;}
```

C♯ 5.0 Async version

```
public static async Task<long> LengthAsync(Stream src) {
  var buf = new byte[0x1000]; int bytesRead;
  long totalRead = 0;
  while ((bytesRead = await src.ReadAsync(buf,0,buf.Length))>0)
    totalRead += bytesRead;
  return totalRead;}
```

"Mr C♯ compiler: please build the callback for me"

"Wow, almost the same—and it's good for me too?"

Async: Statics

```
public static async Task<long> LengthAsync(Stream src) {
  var buf = new byte[0x1000]; int bytesRead;
  long totalRead = 0;
  while ((bytesRead = await src.ReadAsync(buf,0,buf.Length))>0)
    totalRead += bytesRead;
  return totalRead;
}
```

C♯ 5.0 adds two keywords `async` and `await`:

- An `async` method must return a `Task<T>` (or `Task` or `void`);
- ...yet exit by `return` of a `T` (not a `Task<T>`).
- Only `async` methods can contain await expressions.
- If `e` has type `Task<U>` then `await e` has type `U`.
- An async can await task of another (asyncs compose)

Async: Dynamics

```
public static async Task<long> LengthAsync(Stream src) {
  var buf = new byte[0x1000]; int bytesRead;
  long totalRead = 0;
  while ((bytesRead = await src.ReadAsync(buf,0,buf.Length))>0)
    totalRead += bytesRead;
  return totalRead;
}
```

Calling `LengthAsync()`:

- Creates a fresh incomplete task for this call.
- Executes until the next `await` expression.
- If awaited argument is complete now:
  - continue with result (fast path).
  - otherwise, suspend until completion (slow path).
- `return` completes this call's task.
- The first suspend yields the incomplete task to the caller.

Fine points

An async method call:

- runs synchronously until or unless it needs to suspend;
- once suspended, may complete asynchronously;
- if never suspended, completes on same thread;
- does not (in itself) spawn a new thread.

Synchronous on entry avoids context switching. Enables fast path optimizations.

Typically, suspensions resume in thread pool or event loop — not on new threads.

Threading details depends on context (it's complicated).

What's the compiler doing?

- make program counter (pc) explicit as finite state machine.
- one state per await; additional states for control-flow.
- save pc + local values on heap using state-full callback.

What's the compiler doing?

- make program counter (pc) explicit as finite state machine.
- one state per await; additional states for control-flow.
- save pc + local values on heap using state-full callback.

```
public static Task<long> LengthAsync(Stream src) {
  var tcs = new TaskCompletionSource<long>(); // tcs.Task new & incomplete
  var state = 0; TaskAwaiter<int> readAwaiter;
  byte[] buf = null; int bytesRead = 0; long totalRead = 0;
  Action act = null; act = () => {
    while (true) switch (state++) {
      case 0: // entry
        buffer = new byte[0x1000]; totalRead = 0; continue; // goto 1
      case 1: // while loop at await
        readAwaiter = src.ReadAsync(buf, 0, buf.Length).GetAwaiter();
        if (readAwaiter.IsCompleted) continue; // continue from 2
        else { readAwaiter.OnCompleted(act); return; } // suspend at 2
      case 2: // while loop after await
        if ((bytesRead = readAwaiter.GetResult()) > 0) {
           totalRead += bytesRead;
           state = 1; continue; } // goto 1
        else continue; // goto 3
      case 3: // while exit
        tcs.SetResult(totalRead); // complete tcs.Task & "return"
        return; // exit machine
  }}; // end of act delegate
  act(); // start the machine on this thread
  return tcs.Task;
} // on first suspend or exit from machine
```

```
public static Task<long> LengthAsync(Stream src) {
  var tcs = new TaskCompletionSource<long>(); // tcs.Task new & incomplete
  var state = 0; TaskAwaiter<int> readAwaiter;
  byte[] buf = null; int bytesRead = 0; long totalRead = 0;
  Action act = null; act = () => {
    while (true) switch (state++) {
        case 0: // entry
          buffer = new byte[0x1000]; totalRead = 0; continue; // goto 1
        case 1: // while loop at await
          readAwaiter = src.ReadAsync(buf, 0, buf.Length).GetAwaiter();
          if (readAwaiter.IsCompleted) continue; // continue from 2
          else { readAwaiter.OnCompleted(act); return; } // suspend at 2
        case 2: // while loop after await
          if ((bytesRead = readAwaiter.GetResult()) > 0) {
            totalRead += bytesRead;
            state = 1; continue; } // goto 1
          else continue; // goto 3
        case 3: // while exit
          tcs.SetResult(totalRead); // complete tcs.Task & "return"
          return; // exit machine
  }}; // end of act delegate
  act(); // start the machine on this thread
  return tcs.Task;
} // on first suspend or exit from machine
```

Outline

Background and Motivation

Example

Semantics

Concurrency

Extensibility

"Semantics"

Current C♯ 5.0 specs:

- Precise prose describing syntax and typing.
- Example source to source translations (like previous slide).

State machine translation is too low-level — hard to follow, trickier to apply.

Our Aim: to give a precise, high-level operational model for:

- programmers (perhaps)
- compiler writers (hopefully)
- researchers (realistically)

No mention of the finite state machine at all!

For the mathematically inclined…

"Pause 'n Play, Formalizing Asynchronous C♯", ECOOP 2012, Beijing.

Pause 'n' Play: Formalizing Asynchronous C♯

Gavin Bierman[1], Claudio Russo[1], Geoffrey Mainland[1],
Erik Meijer[2], and Mads Torgersen[3]

[1] Microsoft Research
[2] Microsoft Corp. and TU Delft
[3] Microsoft Corporation
{gmb,crusso,gmainlan,emeijer,madst}@microsoft.com

Abstract. Writing applications that connect to external services and yet remain responsive and resource conscious is a difficult task. With the rise of web programming this has become a common problem. The solution lies in using asynchronous operations that separate issuing a request from waiting for its completion. However, doing so in common object-oriented languages is difficult and error prone. Asynchronous operations rely on callbacks, forcing the programmer to cede control. This inversion of control-flow impedes the use of structured control constructs, the staple of sequential code. In this paper, we describe the language support for asynchronous programming in the upcoming version of C♯. The feature enables asynchronous programming using structured control constructs. Our main contribution is a precise mathematical description that is abstract (avoiding descriptions of compiler-generated state machines) and yet sufficiently concrete to allow important implementation properties to be identified and proved correct.

1 Introduction

Mainstream programmers are increasingly adopting asynchronous programming techniques once the preserve of hard-core systems programmers. This adoption is driven by a variety of reasons: hiding the latency of the network in distributed applications; maintaining the responsiveness of single-threaded applications or simply avoiding the resource cost of creating too many threads. To facilitate this programming style, operating systems and platforms have long provided non-blocking, asynchronous alternatives to possibly blocking, synchronous operations. While these have made asynchronous programming possible they have not made it easy.

The basic principle behind these asynchronous APIs is to decompose a synchronous operation that combines issuing the operation with a blocking wait for its completion, into a non-blocking initiation of the operation, that immediately returns control, and some mechanism for describing what to do with the operation's result once it has completed. The latter is typically described by a callback—a method or function. The callback is often supplied with the initiation as an additional argument. Alternatively, the initiation can return a handle which the client can use to selectively register an asynchronous callback or (synchronously) wait for the operation's result.

Whatever the mechanism, the difficulty with using these APIs is fundamentally this: to transform a particular synchronous call-site into an asynchronous call-site requires the programmer to represent the continuation of the original site as a callback. Moreover, for this callback to resume from where the synchronous call previously returned, it must preserve all of the state pertinent to the continuation of the call. Some aspects of the state

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 233–257, 2012.

Take home:

- **async** can be given a "high-level" operational semantics
- as a "simple" extension of C♯ 4.0 semantics
- without resorting to state machines
- correctness proof is subtle

Here's some intuition…

Processes, Stacks and Frames

- A process is a collection of stacks:
  - mutating a shared heap
  - interleaving execution
- A stack (thread) is a sequence of frames (activation records).
- Topmost frame of a stack is active, lower frames are blocked.
- Frames store locals and pc, but come in two flavors:
  - synchronous (as usual)
  - asynchronous referencing a unique task.

Task States

A `task` is a state-full object.

A `Task<T>` represents a computation producing a `T`.

That computation is either:

incomplete: running with a growing list of waiters; or

complete: done with a value v of type `T`.

Task States

A `task` is a state-full object.

A `Task<T>` represents a computation producing a `T`.

That computation is either:

incomplete: running with a growing list of waiters; or

complete: done with a value v of type `T`.

Lifetime of a task:

$$\to \textsf{running}(\epsilon) \to \ldots \textsf{running}(\overline{w}) \to \textsf{running}(w, \overline{w}) \to \ldots \textsf{done}(v)$$

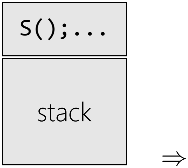
- fresh tasks start off running with no waiters (ϵ).
- waiters may be added to a running task.
- once done, a task never changes state again.

Synchronous calls and returns

- A call to a synchronous method:
  1. pushes a new synchronous frame on the stack
  2. blocks calling frame until return
- Return from a synchronous frame:
  1. pops the active frame
  2. returns value and control to calling frame (now active).

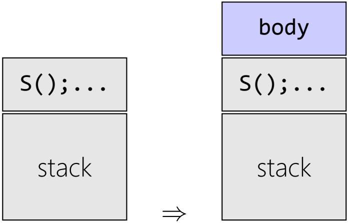
Synchronous call and return (cartoon)

```
T S(){ body };
```



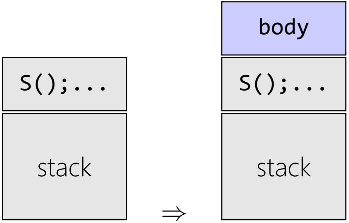
Synchronous call and return (cartoon)

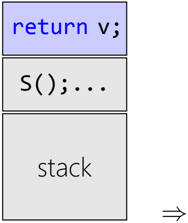
```
T S(){ body };
```



Synchronous call and return (cartoon)

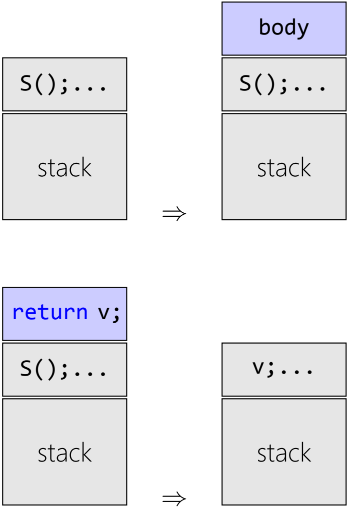
```
T S(){ body };
```





Synchronous call and return (cartoon)

```
T S(){ body };
```

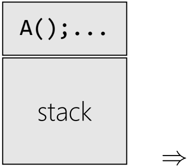


Asynchronous calls and returns

- A call to an asynchronous method:
    1. allocates an incomplete task for that call with no waiters
    2. pushes an asynchronous frame referencing the task
    3. blocks caller until await or return
- return from an asynchronous frame:
    1. stores the return value in its task
    2. resumes the task's waiting frames on fresh stacks
    3. pops the asynchronous frame
    4. remaining stack (if any) proceeds with completed task.

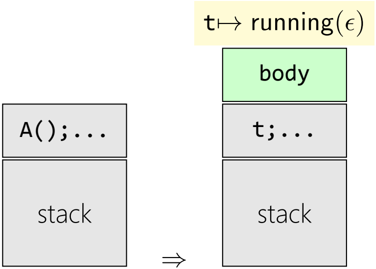
Asynchronous call and return (cartoon)

```
async Task<T> A(){ body };
```



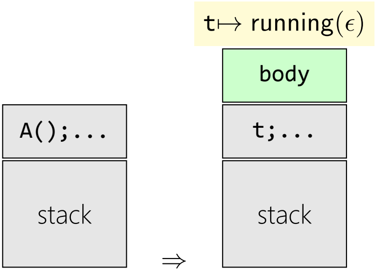
Asynchronous call and return (cartoon)

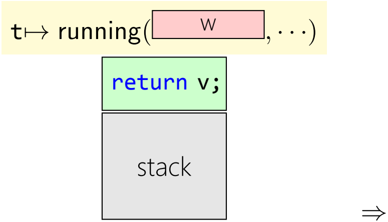
```
async Task<T> A(){ body };
```



Asynchronous call and return (cartoon)

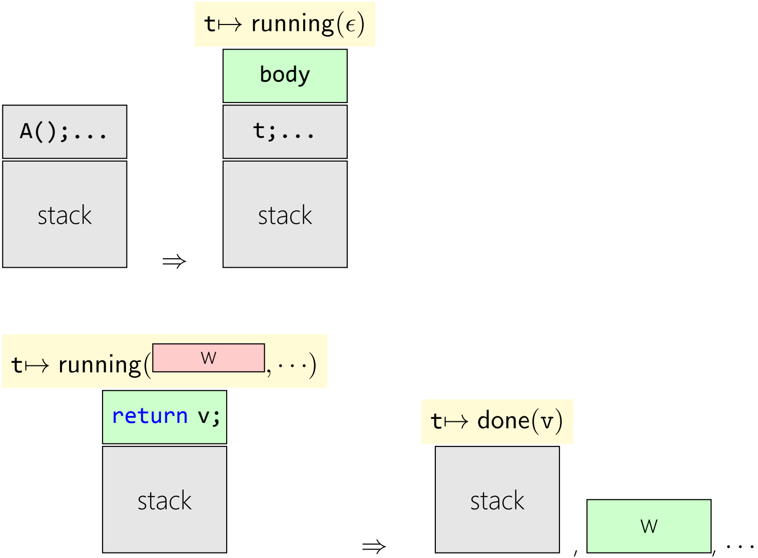
```
async Task<T> A(){ body };
```





Asynchronous call and return (cartoon)

```
async Task<T> A(){ body };
```



Await expressions

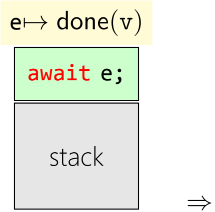
One can only await from an asynchronous method, i.e. an asynchronous frame.

- Await on an completed task with result v:
  1. evaluates to v
  2. proceeds with the active frame
- Await on an incomplete task:
  1. adds the active frame to the task's list of waiters
  2. pops the frame
  3. remaining stack (if any) proceeds with incomplete task

  (active frame is suspended at read of task's result)

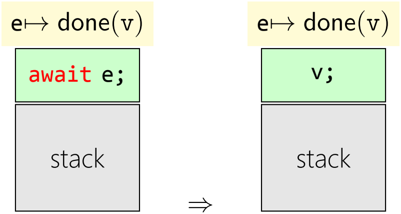
Await expressions (cartoon)

```
await e;
```



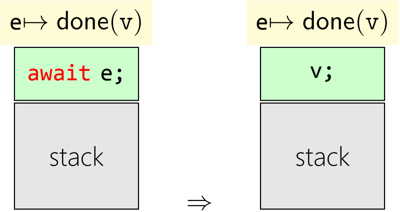
Await expressions (cartoon)

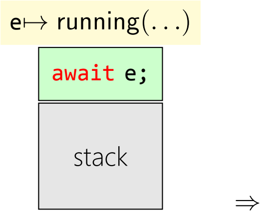
```
await e;
```



Await expressions (cartoon)

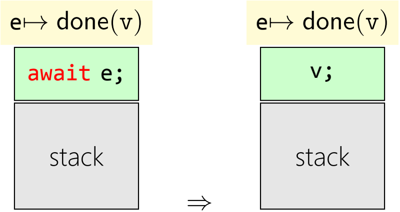
```
await e;
```

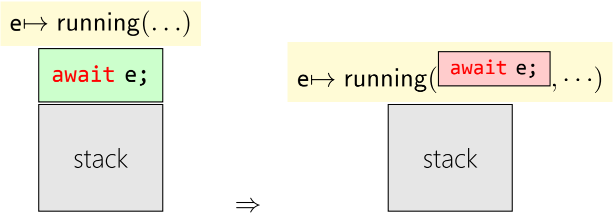




Await expressions (cartoon)

```
await e;
```





Outline

Background and Motivation

Example

Semantics

Concurrency

Extensibility

Synchronous Stream Copy

```
public static long CopyTo(Stream src,Stream dst) {
  var buf = new byte[0x1000];
  int bytesRead;
  long totalRead = 0;
  while ((bytesRead = src.Read(buf, 0, buf.Length)) > 0) {
    dst.Write(buf, 0, bytesRead);
    totalRead += bytesRead;
  }
  return totalRead;
}
```

- copies **`src`** to **`dst`** in chunks (similar to **`Length(src)`**)
- may block at **`src.Read(...)`** and **`dst.Write(...)`**
- synchronous: caller must wait until **`CopyTo`** returns

Aynchronous Stream Copy (Sequential)

```
public static async Task<long> CopyToAsync(Stream src,Stream dst){
  var buf = new byte[0x1000];
  int bytesRead;
  long totalRead = 0;
  while ((bytesRead = await src.ReadAsync(buf, 0, buf.Length)) > 0) {
    await dst.WriteAsync(buf, 0, bytesRead);
    totalRead += bytesRead;
  }
  return totalRead;
}
```

- similar code to synchronous version
- asynchronous: caller proceeds from first incomplete await

Aynchronous Stream Copy (Sequential)

```
public static async Task<long> CopyToAsync(Stream src,Stream dst){
  var buf = new byte[0x1000];
  int bytesRead;
  long totalRead = 0;
  while ((bytesRead = await src.ReadAsync(buf, 0, buf.Length)) > 0) {
    await dst.WriteAsync(buf, 0, bytesRead);
    totalRead += bytesRead;
  }
  return totalRead;
}
```

- similar code to synchronous version
- asynchronous: caller proceeds from first incomplete await

Though asynchronous, this code remains sequential: the next read happens after completion of the previous write.

- no faster than synchronous version (probably slower)

Aynchronous Stream Copy (Sequential)

```
public static async Task<long> CopyToAsync(Stream src,Stream dst){
  var buf = new byte[0x1000];
  int bytesRead;
  long totalRead = 0;
  while ((bytesRead = await src.ReadAsync(buf, 0, buf.Length)) > 0) {
    await dst.WriteAsync(buf, 0, bytesRead);
    totalRead += bytesRead;
  }
  return totalRead;
}
```

- similar code to synchronous version
- asynchronous: caller proceeds from first incomplete await

Though asynchronous, this code remains sequential: the next read happens after completion of the previous write.

- no faster than synchronous version (probably slower)

We can do better...

Aynchronous Stream Copy (Concurrent)

For better performance, we can overlap tasks.

```
public static async Task<long> CopyToConcurrent(Stream src,Stream dst){
  var buf = new byte[0x1000]; var lastbuf = new byte[0x1000];
  int bytesRead; long totalRead = 0; Task lastwrite = null;
  while((bytesRead = await src.ReadAsync(buf,0,buf.Length)) > 0){
    if (lastwrite != null) await lastwrite; // wait later
    lastwrite = dst.WriteAsync(buf, 0, bytesRead); // issue now
    totalRead += bytesRead;
    Swap(ref buf, ref lastbuf);
  }
  if (lastwrite != null) await lastwrite;
  return totalRead;
}
```

- (dynamically) overlaps last write with current read

```
lastwrite = dst.WriteAsync(...);
// issue next read ...
await lastwrite;
```

- exploits separation of task creation and await

Outline

Background and Motivation

Example

Semantics

Concurrency

Extensibility

Awaitable types

Although one typically awaits tasks, any awaitable type will do.

Awaitability is defined by a ”pattern” of methods.

Tasks are awaitable, but other types can be too.

What’s this for?

Awaitable types

Although one typically awaits tasks, any awaitable type will do.

Awaitability is defined by a "pattern" of methods.

Tasks are awaitable, but other types can be too.

What's this for?
Integration with other callback-based synchronization constructs.

Awaitables

An expression `e` is awaitable with type `T` if:
`e.GetAwaiter()` returns some awaiter, `w`, with (non-blocking) members:

- `w.IsCompleted`: testing if `e` is complete now.
- `w.OnCompleted(r)`: registering a callback, `r` (an `Action`).
- `w.GetResult()`: returning a `T` provided `e` is complete.

Proviso:

"Callback *r* must be invoked at most once."

(`r` is a one-shot continuation.)

Await, desugared

```
x = await e;
```

is sugar for (something like)

```
{  var w = e.GetAwaiter();
   if (w.IsCompleted) { goto r;} // fast path
   else {
     w.OnCompleted(() => { goto r; }); // slow path
     suspend;
   };
r: x = w.GetResult();
}
```

(this is pseudo-code, not legal C♯).

An Example: Synchronous Channels

A (swap) channel, `c` of type `Chan<A,B>` provides synchronous message passing between tasks.

A sender calls `c.Send(A a)` to obtain a value of type `B`.

A receiver calls `c.Receive(B b)` to obtain a value of type `A`.

Senders and receivers swap messages.

Senders wait until or unless there is a matching receiver and vice versa.

Channel implementation

Here's an implementation:

```
public class Chan<A,B> {
  readonly Queue<Tuple<A, Action<B>>> SendQ =
     new Queue<Tuple<A, Action<B>>>();
  readonly Queue<Tuple<B, Action<A>>> RecvQ =
     new Queue<Tuple<B, Action<A>>>();
  public SwapAwaiter<B,A> Receive(B b)
   { return new SwapAwaiter<B, A>(b, this, RecvQ, SendQ); }
  public SwapAwaiter<A,B> Send(A a)
   { return new SwapAwaiter<A, B>(a, this, SendQ, RecvQ); }
}
```

A channel `c` stores two (symmetric) queues.

- `SendQ` pairs messages of type `A` with (send) continuations expecting a `B`.
- Calling `c.Send(A a)` returns a new Awaiter<A,B> (next slide).
- `this` is used as lock, protecting the state of both queues.

(`RecvQ` and `c.Receive(B b)` are dual).

Channel Awaiter (extract)

```
public partial class SwapAwaiter<A, B> {
  private B reply;
  public bool IsCompleted { get {
      Monitor.Enter(chan);
      if (RecvQ.Count == 0) return false; // holding lock
      var bk = RecvQ.Dequeue();
      Monitor.Exit(chan);
      reply = bk.Item1;
      TaskEx.Run(() => bk.Item2(message));
      return true; }}
  public void OnCompleted(Action k) {
    SendQ.Enqueue(new Tuple<A,Action<B>>(message, b => { reply = b; k(); }));
    Monitor.Exit(chan); } // release lock acquired by IsCompleted
  public B GetResult() { return reply; }
}
```

Assume message to send stored in field `a`:

- `IsCompleted` locks both channels,
  - If `RecvQ` empty, `IsCompleted` returns false (holding lock).
  - Otherwise, takes `(b,k)` from `RecvQ`, saves `b` in `this`; resumes `k(a);` and returns `true` (releasing lock).
- `OnCompleted(r)`: Given continuation `(`r),
  - makes continuation `k` that sets `b` before continuing like $r()$;
  - enqueues `(a,k)` in `SendQ` (thus "blocking");
  - releases lock.
- `GetResult()` returns saved `b` (guaranteed to be set).

(Some) Related work

[Too many to mention!]

Highly relevant:

- continuations (especially delimited and one-shot).
- C♯ iterators (a.k.a. generators) use similar technology (finite state machines).

  Fun exercise: emulate async with iterators and vice versa!
- F♯'s asynchronous workflows:
  - similar ends; different means and tradeoffs.
  - syntactically heavier (do-notation).
  - workflows are (inert) values: easier to compose first, run later.
  - more general (many-shot continuations).
  - less efficient (every suspend allocates a new continuation).
- Scala's `scala.util.continuations` library:
  - employs novel type-directed selective CPS-transform:
  - syntactically lighter weight than F♯.
  - similar trade-offs to F♯.

[Apologies to other Highly relevant authors☺]

Summary & Conclusions

- C♯ 5.0 makes it much easier to write asynchronous code:
  - No need to roll your own callbacks
  - Builds upon existing `Task` library (which has lots of goodies)
- The essence of these new features can be captured precisely:
  - No mention of finite-state machine encoding
  - Read our paper for details of:
    1. One-shot semantics
    2. Tail-call optimizations
    3. Awaitable patterns
    4. Exceptions

Links

Paper:
Pause 'n' Play: Formalizing Asynchronous C#
G. Bierman, C. Russo, G. Mainland (MSRC), E. Meijer, M. Torgersen (Microsoft Corp.), ECOOP 2012.
`http://rd.springer.com/chapter/10.1007/978-3-642-31057-7_12`

Lang.Next 2012 (video):
Language Support for Asynchronous Programming
Mads Torgersen (C# team)
`http://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2012/`

Resources:
Visual Studio Asynchronous Programming
`http://msdn.microsoft.com/en-us/vstudio/async.aspx`

Questions?