Céu

going *”async”*

Synchronous Model

- Implicit synchronization
  - simpler reasoning about time
- Compositions of activities
  - automatic destruction
- Safe shared memory
  - no locks, no race conditions

Limitations

- No support for
  - Time-consuming operations
    - unbounded loops are refused
    - blocking C calls
  - Input generation
    - would break the synchronous semantics
  - Parallelism
    - scheduler is deterministic
    - no localization of data

Time-consuming operations

```
   input void A, B;
   par/or do
>      await A;
       loop do
          <...>  // some calculation (no awaits)
       end
   with
>      await B;
   end
   <...>          // maybe never reached
```

Input generation

```
   input void A, B;
   par/or do
>       await A;
        emit B;
   with
>       await A;
         <...>          // ?
   with
>       await B;
        <...>          // ?
   end
```

Parallel execution

```
   input void A, B;
   var int x;

   par/or do
>      await A;
       x = 1;      // race condition
   with
>      await A;
       x = 2;      // race condition
   end             // race condition

   _printf("x = %d\n", x);
```

Asynchronous blocks

- Can contain unbounded loops
- Can make blocking C calls
- Can emit input events
- Can be parallelizable
- Two ”flavors”:
  - `async do <...> end`
    - portable, non-parallelizable
  - `async thread do <...> end`
    - non-portable, parallelizable

Asynchronous blocks

```
var int ret;
loop do
    <...>
end
return ret;
```

```
var int ret =
    async do
        var int v;
        loop do
            <...>
        end
        return v;
    end;
return ret;
```

- Understandable as the sequence "*emit*+*await*":

```
emit OUT_i(n);

ret=await IN_i;
```

Asynchronous blocks

```
input int A;
<...>
par do
     async do
          emit A => 1;
     end
with
     var int v = await A;
     return v;
end
```

- Synchronous code always has higher priority

Simple `async`

- Shares code+memory with the synchonous side
- Unbounded loops
  - inserts a check before every loop iteration
- Good
  - portable
  - still no need for synchr
- Bad
  - no real parallelism
  - still no C blocking calls

- ✔ Can contain unbounded loops
- ✗ Can make blocking C calls
- ✔ **Can emit input events**
- ✗ Can be parallelizable

Threaded `async`

- Separate code+memory
  - (i.e. a separate C function with locals)
- Executes in a different thread
- Special **sync** syntax for synchronization
  - (i.e. locks a global shar
- *Parallelism* and *C blockir*

- ✔ Can contain unbounded loops
- ✔ Can make blocking C calls
- ✔ Can emit input events
- ✔ Can be parallelizable

Parallelism example

```
// pthreads

 main() {
    pthread_t t1, t2;
    pthread_create(&t1, calc);
    pthread_create(&t2, calc);
    pthread_join(t1);
    pthread_join(t2);
    exit(0);
 }

 void* calc (void *ptr) {
   int ret, i, j;
   ret = 0;
   for (i=0; i<50000; i++) {
     for (j=0; j<50000; j++) {
       ret = ret + i + j;
     }
   }
   printf("ret = %d\n", ret);
 }

# 17.39s user
#  0.01s system
# 175% cpu 9.890 total
```

```
// ceu

 var int  v1, v2;
 var int* p1 = &v1;
 var int* p2 = &v2;

 par/and do
   async thread (p1) do
     var int ret = _calc();
     sync do
       *p1 = ret;
     end
   end
 with
   async thread (p2) do
     var int ret = _calc();
     sync do
       *p2 = ret;
     end
   end
 end

# 17.38s user
#  0.02s system
# 175% cpu 9.937 total
```

Killing a thread

```
par/or do
    await 1s;
with
    async do
        ...
        sync do
            // side effects
        end
    end
end
```

- **sync** ensures that the thread is still alive

Summary

- Synchronous code (*safety*)
  - determinism
    - sequential scheduling
  - bounded execution
    - no loops
- Asynchronous code (*flexibility*)
  - parallelism
  - blocking calls
  - requires synchronization
    - **emit** and **sync**