Department of Computer Science
University of Bristol

COMS20001 - Concurrent Computing

www.cs.bris.ac.uk/Teaching/Resources/COMS20001

Lecture 12

Execution Control in xC

Sion Hannuna | hannuna@cs.bris.ac.uk
Tilo Burghardt | tilo@cs.bris.ac.uk
Dan Page | daniel.page@bristol.ac.uk

13 Slides

Recap: Channel Arrays

```
//channelarray.xc
#include <platform.h>
#include <stdio.h>

void taskA(chanend c[n], unsigned n) {
  int serving = 1;
  while (serving)
    select {
      case c[int j] :> int data:
        printf("channel %d receives %d\n",j,data);
        c[j] <: data;
        if (data == 0) serving = 0;
        break;
} }

void taskB(chanend c, chanend d, int terminate) {
  int data;
  c <: 1;
  c :> data;
  c <: 2;
  c :> data;
  if (terminate == 1) {
      d :> data;
      c <: 0;
      c :> data;
  } else d <: 0;
}

int main() {
  chan c[2],d;
  par {
    on tile[0]: taskA(c,2);
    on tile[1]: taskB(c[0],d,1);
    on tile[1]: taskB(c[1],d,0);
  }
  return 0;
}
```

Recap: Interface Arrays

```
//interfacearrays.xc
#include <platform.h>
#include <stdio.h>
#include <string.h>

//define a communication interface i
typedef interface i {
  int f(int a[]);
  void g();
} i;

//server task providing functionality of i
void myServer(server i myInterface[n], unsigned n) {
  int serving = 1;
  int data[2] = {10,11};
  while (serving)
    select {
      case myInterface[int j].f(int a[]) -> int returnval:
        printf("f called from %d \n",j);
        memcpy(a, data , 2*sizeof(int));
        returnval = a[0];
        break;
      case myInterface[int j].g():
        printf("g was called from %d\n",j);
        serving = 0;
        break;
} }

//client task calling functions
void myClient(client i myInterface, int j) {
  int a[2] = {0,0};
  printf("f returns: %d \n", myInterface.f(a));
  printf("a set to: [%d,%d] \n", a[0], a[1]);
  if (j==1) myInterface.g();
}

//main starting two threads
int main() {
  interface i myInterface[2];
  par {
    on tile[0]: myServer(myInterface,2);
    on tile[1]: {
                  myClient(myInterface[0],0);
                  myClient(myInterface[1],1);
                }
  }
  return 0;
}
```

Recap: Multiple Interfaces – One Server `select`

```
//interface2.xc
#include <platform.h>
#include <stdio.h>

//define a communication interface i
typedef interface i {
  void f(int x);
  void g();
} i;

//server task providing functionality of two IFs
void myServer(server i myInterface,
              server i myInterface2) {
  int serving = 1;
  while (serving)
    select { //SINGLE select statement for two IFs!!!
      case myInterface2.f(int x):
        printf("f got data in IF2: %d \n", x);
        break;
      case myInterface2.g():
        printf("g was called in IF2\n");
        break;
      case myInterface.f(int x):
        printf("f got data in IF1: %d \n", x);
        break;
      case myInterface.g():
        printf("g was called in IF1\n");
        serving = 0;
        break;
  }
} ...
```

```
...
//client task calling function of task 2
void myClient(client i myInterface,
              client i myInterface2) {
  myInterface.f(2);
  myInterface.f(1);
  myInterface2.f(3);
  myInterface2.f(4);
  myInterface2.g();
  myInterface.g();
}

//main starting two threads calling
//over two interfaces
int main() {
  interface i myInterface;
  interface i myInterface2;
  par {
    on tile[0]: myServer(myInterface,
                         myInterface2);
    on tile[1]: myClient(myInterface,
                         myInterface2);
  }
  return 0;
}
```

```
f got data in IF1: 2
f got data in IF1: 1
f got data in IF2: 3
f got data in IF2: 4
g was called in IF2
g was called in IF1
```

Sharing Cores

```
//combinable.xc
#include <platform.h>
#include <stdio.h>

[[combinable]] //allow function to SHARE a logical core
void taskA(chanend c[n], unsigned index, unsigned n) {
  while (1)
    select {
      case c[int j] :> int data:
        printf("channel %d%d gets %d\n",index,j,data);
        c[j] <: data;
        break;
} }

void taskB(chanend c, int index) {
  int data;
  c <: index;
  c :> data;
}

int main() {
  chan c[2],d[2];
  par {
    on tile[0].core[0]: taskA(c,1,2); //share core 0
    on tile[0].core[0]: taskA(d,0,2); //share core 0
    on tile[1]: taskB(c[0],1);
    on tile[1]: taskB(c[1],2);
    on tile[1]: taskB(d[0],3);
    on tile[1]: taskB(d[1],4);
  }
  return 0;
}
```

instructs compiler to merge **select** statements of all combined functions per core into single **select** statement

function needs to be of predefined layout to qualify for the **combinable** option:

1) must return **void**

2) last statement needs to be a **while(1)** loop encapsulating a single **select** statement

single logical core specification to run both instances of taskA

Further Details on Combinable Functions

- no channels allowed between combinable threads
- alternative to core specification is use of **`[[combine]]`**:

```
void f() {
  [[combine]]
  par {
    taskA (" task1 ");
    taskB (" task2 ");
} }
```

- interfaces need to be used to share data between threads running on the same logical core
- combinable functions can naturally be combined
- the compiler will produce an error if non-combinable functions are placed on the same core

Distributables

```
//distributable.xc
#include <platform.h>
#include <stdio.h>
#include <string.h>

//define a communication interface i
typedef interface i {
  int f(int a[]); void g(); } i;

//server tasks providing functionality are ONLY responsive
[[distributable]] //allows on-demand core allocation to caller core
void myServer(server i myInterface[n], unsigned n, int index) {
  while (1)
    select {
      case myInterface[int j].f(int a[]) -> int returnval:
        printf("f called from i%d-c%d \n",index,j);
        returnval = a[j]+j;
        break;
      case myInterface[int j].g():
        printf("g was called from i%d-c%d\n",index,j);
        break;
} }

//client task calling functions
void myClient(client i myInterface, int j) {
  int a[2] = {1+j,2+j};
  printf("Client %d received %d\n",j,myInterface.f(a));
  if (j%2==0) myInterface.g();
}

//main starting six threads
int main() {
  interface i myInterface[2];
  interface i myInterface1[2];
  par { //uses only 4 cores
    myClient(myInterface[0],0);
    myClient(myInterface[1],1);
    myClient(myInterface1[0],2);
    myClient(myInterface1[1],3);
    myServer(myInterface,2,0);
    myServer(myInterface1,2,1);
  }
  return 0;
}
```

```
f called from i1-c1
f called from i0-c1
f called from i1-c0
f called from i0-c0
Client 3 received 6
Client 1 received 4
Client 2 received 3
Client 0 received 1
g was called from i1-c0
g was called from i0-c0
```

instructs compiler only to allocate core in case function is triggered/called (core of caller can be utilised for this, if on same tile)

Distributables must be:

1) functions that satisfy combinable criteria
2) cases within the **select** only respond to interface calls

Further Details on Distributable Functions

- if on the same tile with caller, distributables are not allocated a core
- instead, the context on the caller's core will be swapped to carry out the triggered case in the **`select`** statement
- after the function is completed (at **`break`**), the context is swapped back to the caller
- if distributables are connected across tiles then they will run as normal tasks on a core (though they are still combinable functions)
- if a distributable is connected to several tasks, these cannot change the distributable's state concurrently
- in this case the compiler implicitly locks the distributable to protect the state of the task from concurrent access
- the distributable is then treated as a critical section of code
- interfaces need to be used to share data between threads running on the same logical core

More Details on Timers

- every tile has a reference clock ticking at 100MHz
- each reference clock has
  an associated 32-bit counter
  → one can thus measure up to
  $2^{32} - 1$ ticks (approx 42 sec)
- timers can be used to control
  periodic events combined
  with triggered events
- this is particularly
  resource-efficient in
  combinable functions

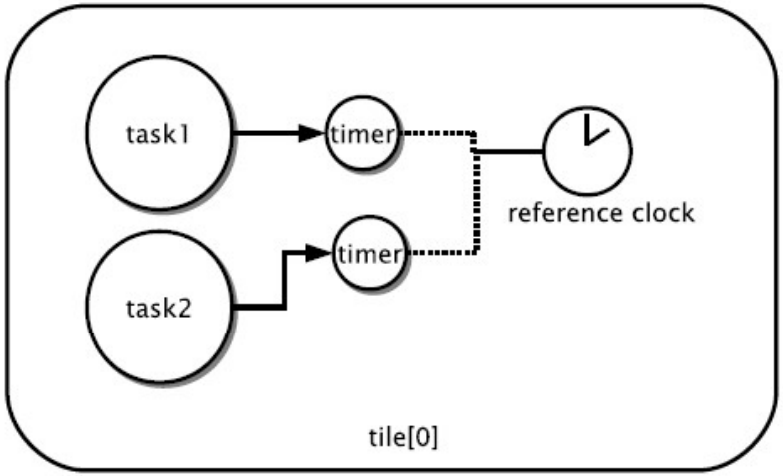

Illustration taken from xC Programming Guide

(courtesy of XMOS)

Periodic Function Calls

```
//periodic.xc
#include <platform.h>
#include <stdio.h>
#include <string.h>

//define a communication interface i
typedef interface i {
  void g();
} i;

//server tasks providing functionality are ONLY responsive
void myServer(server i myInterface[n], unsigned n) {
  while (1)
    select {
      case myInterface[int j].g():
        printf("g was called from channel %d\n",j);
        break;
} }

//client task calling functions
void myClient(client i myInterface, int j) {
  timer t;
  unsigned int time;
  const unsigned int period = 100000000; // period of 1s
  t :> time; // get the initial timer value
  while (1) {
    select {
      case t when timerafter ( time ) :> void :
        myInterface.g(); // perform periodic task
        time += period ;
        break ;
} } }

//main starting three threads
int main() {
  interface i myInterface[2];
  par {
    on tile[0]:myClient(myInterface[0],0);
    on tile[1]:myClient(myInterface[1],1);
    on tile[1]:myServer(myInterface,2);
  }
  return 0;
}
```

There are two instances of the client task created.

Each of them utilises a timer variable.

In this particular program each variable reads its values from a different timer – one located on **tile[0]** the other one located on **tile[1]**.

```
g was called from channel 0
g was called from channel 1
g was called from channel 0
g was called from channel 1
g was called from channel 0
g was called from channel 1
...
```

Guarding a Case

```
//guards.xc
#include <platform.h>
#include <stdio.h>
#include <string.h>

//define a communication interface i
typedef interface i {
  void g();
} i;

//server tasks providing functionality are ONLY responsive
void myServer(server i myInterface[n], unsigned n) {
  while (1)
    select {
      case myInterface[int j].g():
        printf("g was called from channel %d\n",j);
        break;
} }

//client task calling functions
void myClient(client i myInterface, int enabled) {
  timer t;
  unsigned int time;
  const unsigned int period = 100000000; // period of 1s
  t :> time; // get the initial timer value
  while (1) {
    select {
      case enabled => t when timerafter ( time ) :> void :
        myInterface.g(); // perform periodic task
        time += period ;
        break ;
} } }

//main starting three threads
int main() {
  interface i myInterface[2];
  par {
    on tile[0]:myClient(myInterface[0],0);
    on tile[1]:myClient(myInterface[1],1);
    on tile[1]:myServer(myInterface,2);
  }
  return 0;
}
```

One can conditionally enable/disable certain cases of a **select** statement.

The code will, thus, only react to triggers if the guard expression in front of them is evaluated to non-zero.

```
g was called from channel 1
g was called from channel 1
g was called from channel 1
g was called from channel 1
g was called from channel 1
g was called from channel 1
...
```

Guarding an Interface

```
//guardif.xc
#include <platform.h>
#include <stdio.h>
#include <string.h>

//define a communication interface i
typedef interface i {
  [[guarded]] void g();
} i;

//server tasks providing functionality are ONLY responsive
void myServer(server i myInterface[n], unsigned n, unsigned enabled) {
  while (1)
    select {
      case 1-enabled => myInterface[0].g():
        printf("g was called from channel 0\n");
        break;
      case enabled => myInterface[1].g():
        printf("g was called from channel 1\n");
        break;
} }

//client task calling functions
void myClient(client i myInterface) {
  timer t;
  unsigned int time;
  const unsigned int period = 100000000; // period of 1s
  t :> time; // get the initial timer value
  while (1) {
    select {
      case t when timerafter ( time ) :> void :
        myInterface.g(); // perform periodic task
        time += period ;
        break ;
} } }

int main() {//main starting three threads
  interface i myInterface[2];
  par {
    on tile[0]:myClient(myInterface[0]);
    on tile[1]:myClient(myInterface[1]);
    on tile[1]:myServer(myInterface,2,1);
  }
  return 0;
}
```

If an interface function is guarded **anywhere** in the program, it must be marked as possibly guarded in the interface declaration using the **[[guarded]]** attribute.

```
g was called from channel 1
g was called from channel 1
g was called from channel 1
g was called from channel 1
g was called from channel 1
g was called from channel 1
...
```

Order Enforcement

```
//ordering.xc
#include <platform.h>
#include <stdio.h>
#include <string.h>

//define a communication interface i
typedef interface i {
  void g();
} i;

//server tasks providing functionality are ONLY responsive
void myServer(server i myInterface[n], unsigned n) {
  timer t;
  while (1) {
    [[ordered]] //guarantees that interface is executed
    select {
      case myInterface[int j].g():
        printf("accept from channel %d\n",j); break;
      case t :> void:
        printf("timer run\n");  break;
} } }

//client task calling functions
void myClient(client i myInterface) {
  timer t;
  unsigned int time;
  const unsigned int period = 100000000;
  t :> time; // get the initial timer value
  while (1) {
    select {
      case t when timerafter ( time ) :> void :
        myInterface.g(); // perform periodic task
        time += period ; break ;
} } }

//main starting three threads
int main() {
  interface i myInterface[2];
  par {
    on tile[0]:myClient(myInterface[0]);
    on tile[1]:myClient(myInterface[1]);
    on tile[1]:myServer(myInterface,2);
  }
  return 0;
}
```

Generally, there is no priority on the cases of a **select**. The select order amongst possible selections is unspecified.

Using **[[ordered]]**, events are ordered in priority from highest to lowest. (Combinables and distributables cannot be ordered.)

In the example that guarantees that the Interface calls are actually answered.

```
accept from channel 0
accept from channel 1
timer run
timer run
timer run
...
```

Select Functions

```
//selects.xc
#include <platform.h>
#include <stdio.h>
#include <string.h>
//define a communication interface i
typedef interface i { void g(); } i;

//reusable case statements
select timerSelect(timer t) {
  case t :> void: printf("timer run\n"); break; }

//server tasks providing functionality are ONLY responsive
void myServer(server i myInterface[n], unsigned n) {
  timer t;
  while (1) {
    [[ordered]] //guarantees that interface is executed
    select {
      case myInterface[int j].g():
        printf("accept from channel %d\n",j); break;
      case timerSelect(t);
} } }

//client task calling functions
void myClient(client i myInterface) {
  timer t, t2;
  unsigned int time;
  const unsigned int period = 100000000;
  t :> time; // get the initial timer value
  while (1) {
    [[ordered]]
    select {
      case t when timerafter ( time ) :> void :
        myInterface.g(); // perform periodic task
        time += period ; break ;
      case timerSelect(t2);
} } }

int main() {//main starting three threads
  interface i myInterface[2];
  par {
    on tile[0]:myClient(myInterface[0]);
    on tile[1]:myClient(myInterface[1]);
    on tile[1]:myServer(myInterface,2);
  }
  return 0;
}
```

One can place one or more **select** cases into a select function.

This allows the programmer to abstract and reuse parts of a **select**.

```
accept from channel 0
timer run
accept from channel 1
timer run
timer run
...
```