CSc 520

Principles of Programming Languages

32 : Control Structures — Coroutines

Christian Collberg

collberg+520@gmail.com

Department of Computer Science

University of Arizona

Coroutines

- Coroutines are supported by Simula and Modula-2. They are similar to Java's threads, except the programmer has to explicitly transfer control from one execution context to another.
- Thus, like threads several coroutines can exist simultaneously but unlike threads there is no central scheduler that decides which coroutine should run next.
- A coroutine is represented by a closure.
- A special operation transfer(C) shifts control to the coroutine C, at the location where C last left off.

Coroutines vs. threads

- Coroutines are like threads, except that with a coroutine you have to explicitly say when you want to transfer from one “process” to another.
- If you have access to coroutines, you can build a thread library on top of it.
- A thread library makes sure that each process gets its fair share of the CPU. With coroutines we’re forced to handle this ourselves, by making sure that we transfer control “often enough” to all coroutines.

Example

- The next slide shows an example from Scott where two coroutines execute “concurrently”, by explicitly transferring control between each other.
- In the example one coroutine displays a moving screen-saver, the other walks the file-system to check for corrupt files.

Example...

```
var us, cfs: coroutine;

coroutine update_screen() {
   ...
   detach
   loop {
      ... transfer(cfs) ...
   }
}

coroutine check_file_system() {  ... }

main () { ... }
```

Coroutines...

```
coroutine check_file_system() {
   ...
   detach
   for all files do {
      ... transfer(cfs)
      ... transfer(cfs)
         ... transfer(cfs) ...
   }
}

main () {
   us := new update_screen();
   cfs := new check_file_system();
   transfer(us);
}
```

Coroutines in Modula-2

Coroutines in Modula-2

- Modula-2's system module provides two functions to create and transfer between coroutines:

```
PROCEDURE NEWPROCESS(
   proc: PROC;                      (* The procedure     *)
   addr: ADDRESS;                   (* The stack         *)
   size: CARDINAL;                  (* The stack size    *)
   VAR new: ADDRESS);               (* The coroutine     *)
PROCEDURE TRANSFER(
   VAR source: ADDRESS;             (* Current coroutine *)
   VAR destination: ADDRESS);       (* New coroutine     *)
```

- The first time TRANSFER is called `source` will be instantiated to the main (outermost) coroutine.

Coroutines in Modula-2...

```
VAR crparams: CoroutineParameters;
    source: ADDRESS; (* current coroutine is called by this *)
    newcr: ADDRESS; (* coroutine just created by NEWPROCESS *)

PROCEDURE Coroutine;
   VAR myparams: CoroutineParameters;
BEGIN
   myparams := crparams;
   TRANSFER(newcr, source); (* return to calling coroutine *)
   (* rest of coroutine *)
END Coroutine;

PROCEDURE Setup(params: CoroutineParameters; proc: PROC);
BEGIN
   NEWPROCESS(proc, addr, size, newcr);
   crparams := params; TRANSFER(source, newcr);
END Setup;
```


[9]

Coroutines in Ruby

Coroutines in Ruby

- Ruby doesn't have co-routines per-se, but Marc De Scheemaecker has a simple library that we can use.

```
class Coroutine
   # Creates a coroutine. The associated block
   # does not run yet.
   def initialize(&block)

   # Starts the block. It's an error to call
   # this method on a coroutine that has
   # already been started.
   def start

   # Switches context to another coroutine. You
   # need to call this method on the current coroutine.
  def switch(coroutine)
```

Coroutines in Ruby

```
    ...

    # Returns true if the associated block is
    # started and has not yet finished
    def running?

    # Returns true if the associated block is
    # finished
  def finished?
end
```

Example

- c1 prints all letters, c2 prints the numbers $1 \ldots 26$. After printing a letter c1 swithces to c2, and vice versa.

```
$c1 = Coroutine::new do
  for i in ’a’..’z’ do
    printf "%s ", i
    $c1.switch($c2)
  end
end

$c2 = Coroutine::new do
  for i in 1..26 do
    printf "%i ", i
    $c2.switch($c1)
  end
end
```

Example. . .

Running the example:

```
$c1.start
printf "\n"
```

yields the result

```
a 1 b 2 c 3 d 4 e 5 f 6 g 7 h 8 i 9 j 10
k 11 l 12 m 13 n 14 o 15 p 16 q 17 r 18
s 19 t 20 u 21 v 22 w 23 x 24 y 25 z 26
```

Iterators using coroutines

- If you have coroutines, implementing iterators becomes trivial! You need one coroutine to generate the values, and another as the main loop.
- Here, the `iterate` function creates a coroutine that generates the elements of an array. For simplicity, we store the result in a global variable `$result`:

```
$result = 0
def iterate(arr,other)
   c = Coroutine::new do
      i = 0
      while i < arr.length
         $result = arr[i]
         i += 1
         c.switch(other)
      end
```

Iterators using coroutines. . .

- Here's the main routine. It creates a coroutine, calls `iterate` to create the iterator coroutine, and then switches back and forth until the iterator is done:

```
main = Coroutine::new do
   a = [1,2,3]
   iter = iterate(a,main)
   while not iter.finished?
      main.switch(iter)
      printf "%i ", $result
   end
   printf "\n"
end
```

- (This code is buggy. Please fix it for me!)

Implementing coroutines

Implementing coroutines

- Each coroutine needs its own stack.
- Each coroutine is represented by a *context block*. In our implementation, the context block contains only one value: the coroutine's stack pointer.

Implementing coroutines. . .

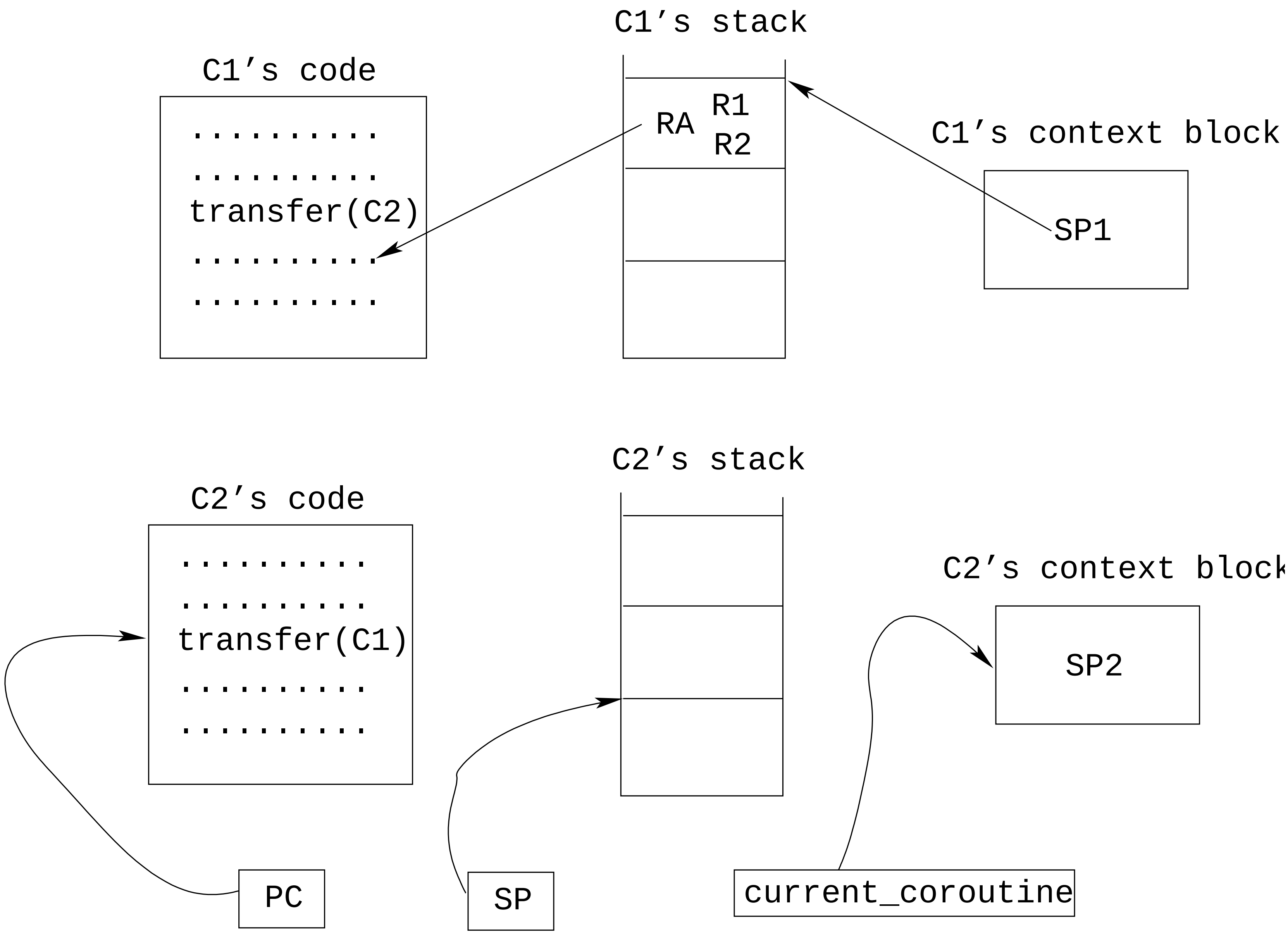
- When coroutine `C2` issues a `transfer(C1)`, the following happens:
  1. `transfer` pushes all registers (including the return address RA on `C2`'s stack.
  2. `transfer` saves the current stackpointer SP into `C2`'s context block.
  3. `transfer` sets `current_coroutine` to `C1`.
  4. `transfer` sets SP to the stackpointer that was saved in `C1`'s context block.
  5. `transfer` pops all saved registers off of `C1`'s stack, including the old return address, RA.
  6. `transfer` does a normal procedure return, which has the effect of setting PC to RA, the location where `C1` wants to continue executing.

Implementing coroutines. . .

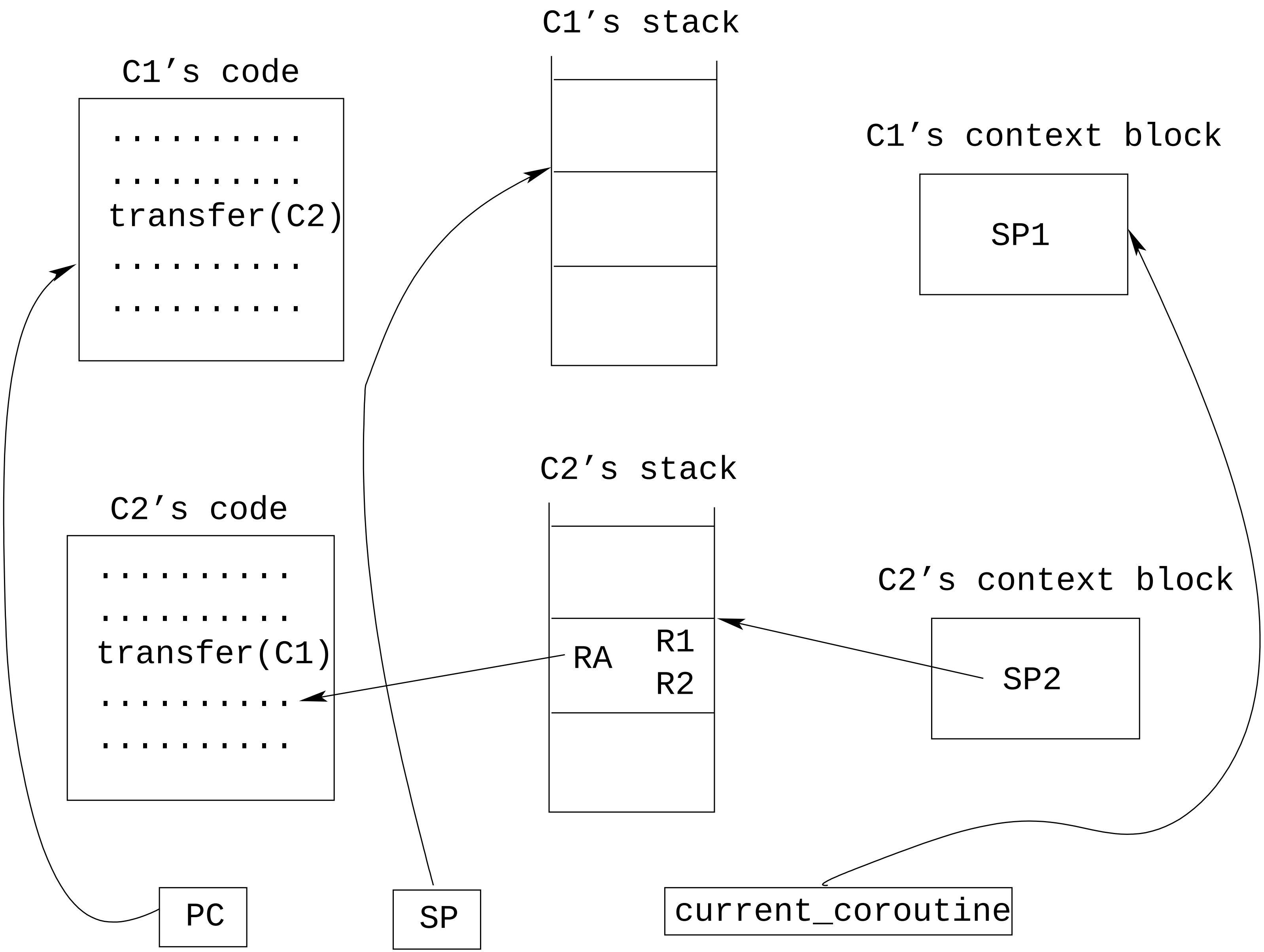
```
current_coroutine := ...
PC := ...
SP :=

transfer(C) {
   push R1,R2,RA
   *current_coroutine := SP
   current_coroutine := C
   SP := *C
   pop R1,R2,RA
   return
}
```

Step 1: Coroutine C2 is running



Step 2: Control is transferred to C2



Readings and References

- Read Scott, pp. 453–459
-
  http://www.mathematik.uni-ulm.de/oberon/0.5/articles/coroutines.html
- The Ruby coroutine package is due to Marc De Scheemaecker, http://liber.sourceforge.net/coroutines.rb.