Tanker om Quasiparalell Sequencing

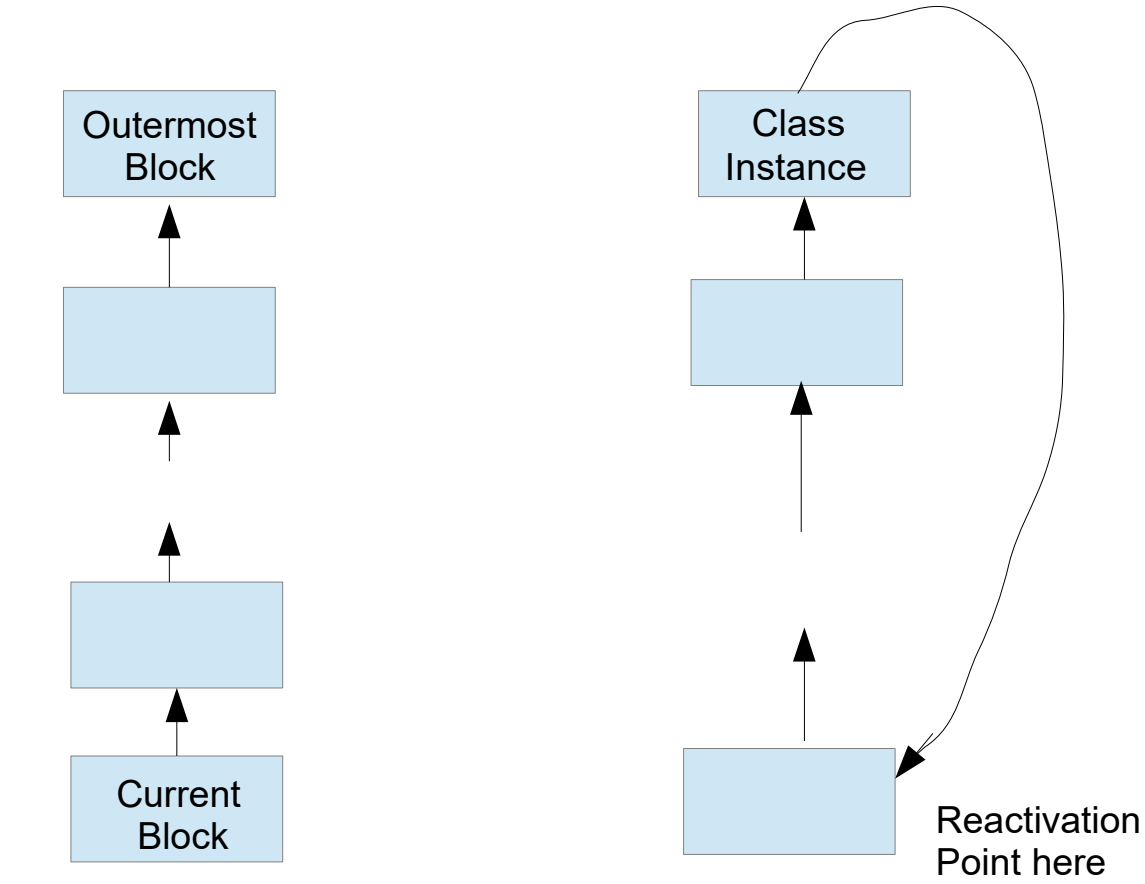

Fig 1. Klassisk illustrering av Detach

Dette dokumentet inneholder tegninger og tanker jeg gjorde meg mens jeg forsøkte å finne en løsning på problemet:

Hvordan implementere Detach/Call/Resume ?

Jeg tenkte at hvis jeg tegnet det opp så ville jeg kanskje komme på en løsning – og det gjorde jeg !

… god lesning.

Litt bakgrunn:

Hva JVM-spec'n sier om Java Heap and Stack(s)

2.5.3 Heap
The Java Virtual Machine has a *heap* that is shared among all Java Virtual Machine threads. The heap is the run-time data area from which memory for all class instances and arrays is allocated.

Altså; Alle Java-class Objekter allokeres på en Heap som ryddes av en Garbage Collector slik vi er vant til.

Alle slags Simula-blokker (class,procedure,...) blir Java-classer og allokeres på Heap'en

Men; Deklarasjons- og Statement-koder er lokale methoder og allokeres på Stack'en Dvs. Eventuelle mellom-resultater finner vi på Stack'en.

2.5.2 Java Virtual Machine Stacks
Each Java Virtual Machine thread has a private *Java Virtual Machine stack*, created at the same time as the thread. A Java Virtual Machine stack stores frames (§2.6).

Altså; Hver tråd har en Stack. Metode-kall genererer 'Frames' som allokeres på Stacken.

2.6 Frames
A *frame* is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions.
A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes, whether that completion is normal or abrupt (it throws an uncaught exception). Frames are allocated from the Java Virtual Machine stack (§2.5.2) of the thread creating the frame. Each frame has its own array of local variables (§2.6.1), its own operand stack (§2.6.2), and a reference to the runtime constant pool (§2.5.5) of the class of the current method.

For å illustrere problematiken ved save/restore av operand-stakken under utføring QPS/Detach I JVM.

Anta følgende Simula Program:

```
System begin ref(C)  x;

    procedure P;
    begin  x:-new C();
              …
              call(x);
    end;

    class Component;
    begin
        procedure Q;
        begin ...  detach;  ...  end;
        ...
        Q;
        ...
    end Component;
    ...
    P;
    ...
end System;
```

De neste par sider kan blas raskt I fram og tilbake. Du får da en animering av effekten av Detach slik jeg først tenkte meg å Implementere det hele.

Legg merke til at Stack-Framen til Q blir frigjort mens framen til C blir hengende og sansynligvis bryter reglene for bruk av stacken.

Det fins løsning på dette ved at ikke bare STM-koden til Q termineres Men hele stacken tilbake til nærmeste block som er et komponent hode, I vårt tilfelle instansen av klassen Component.

Dette blir så pirkete og komplisert at jeg foreslår å vente og se hva 'Project Loom' kommer opp med.

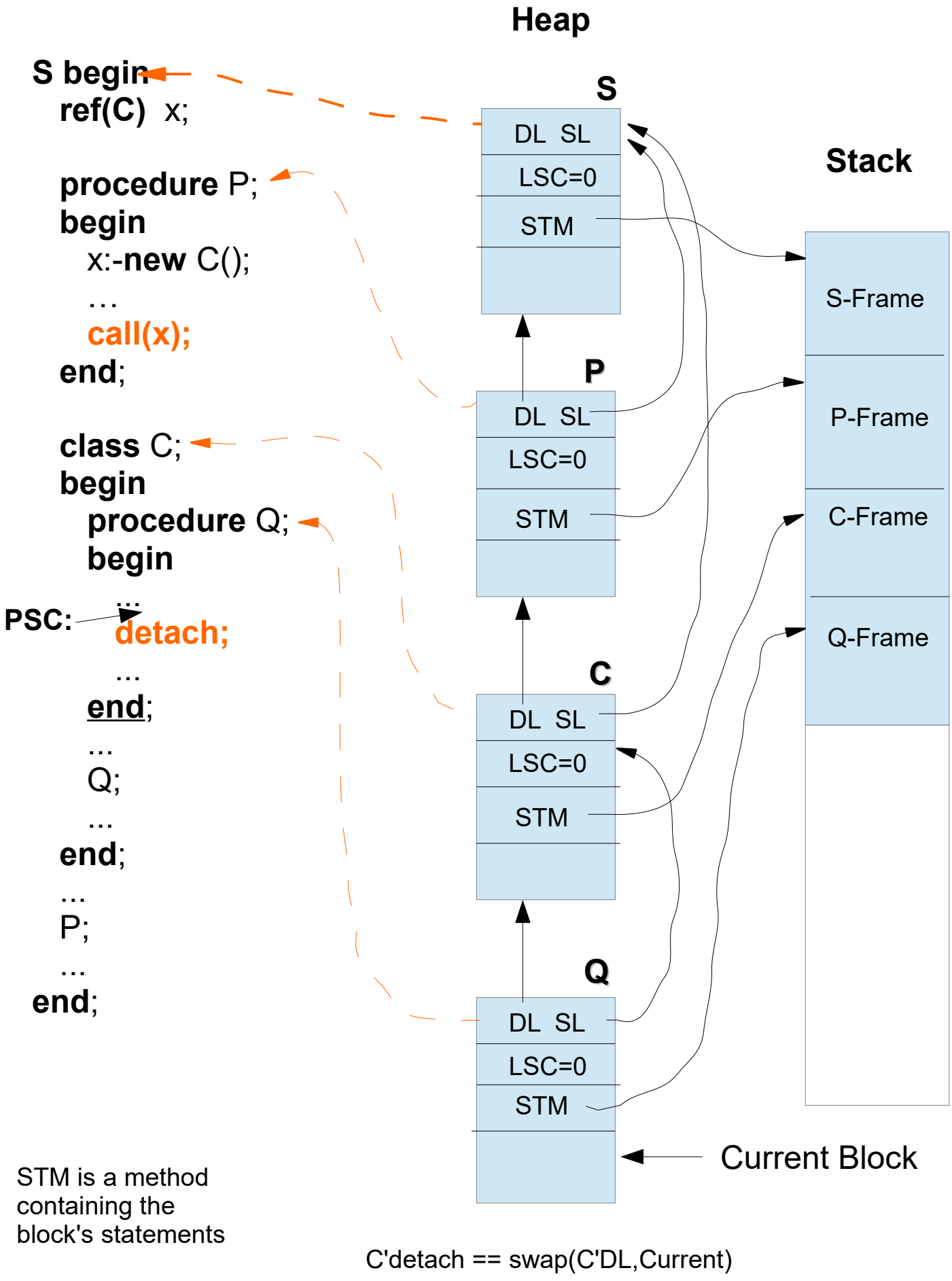

Fig. 2 Just before Detach

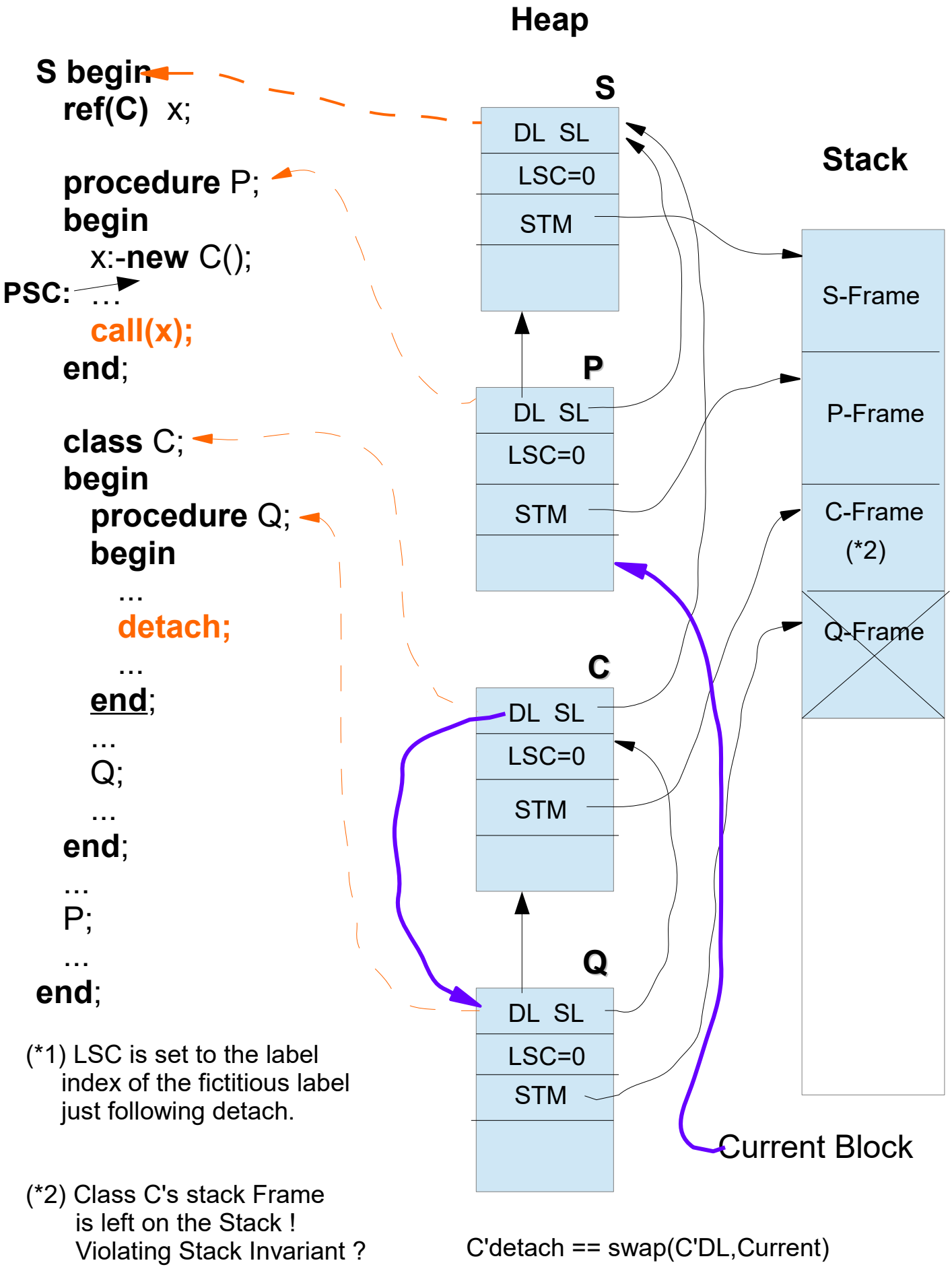

Fig. 3 Just after Detach

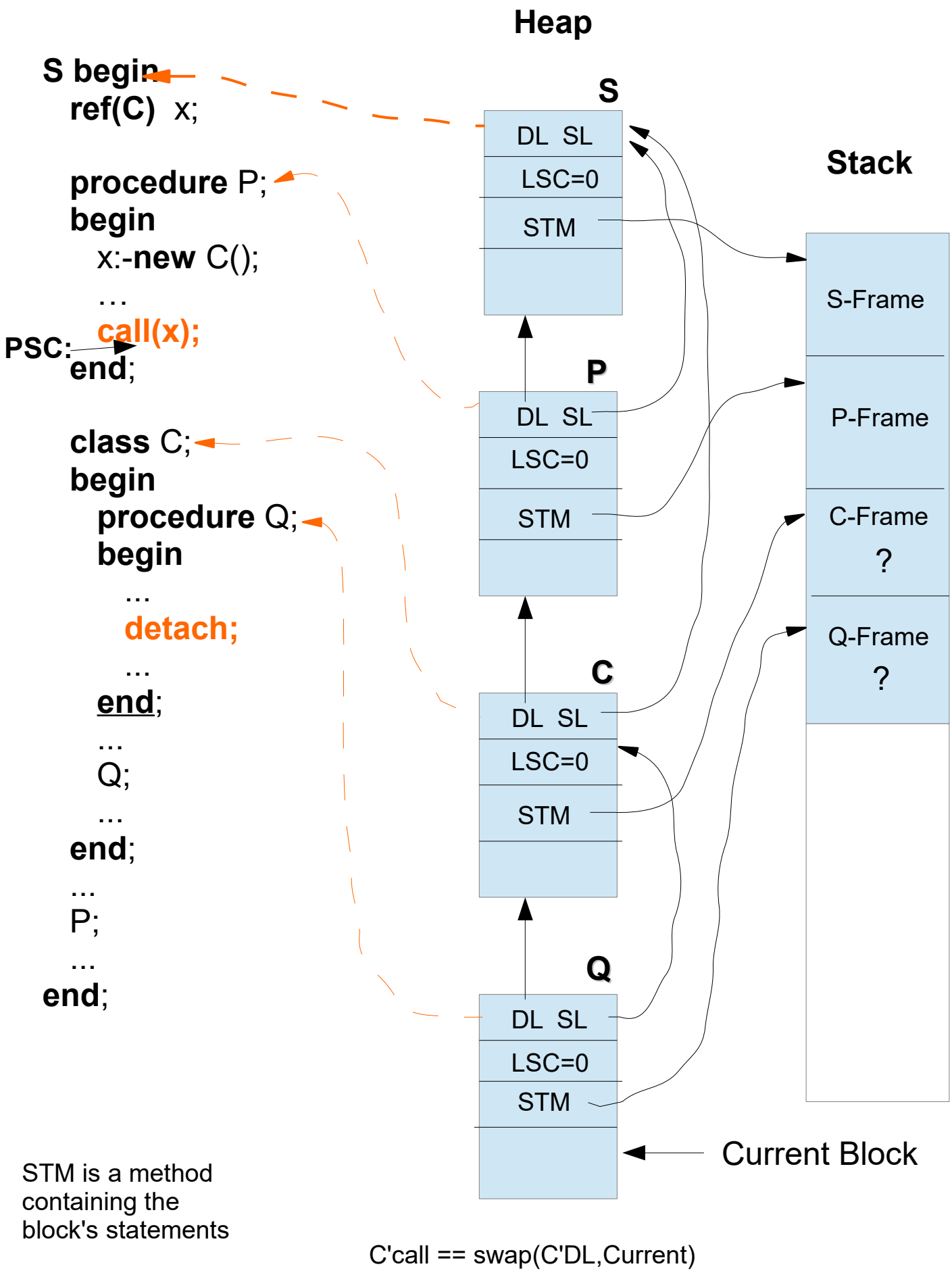

Fig. 4 Just after Call

Project Loom: Fibers and Continuations for the Java Virtual Machine

Everywhere in Project Loom, the word continuation will mean a delimited continuation (also sometimes called a coroutine). Here we will think of delimited continuations as sequential code that may suspend (itself) and resume (be resumed by a caller). Some may be more familiar with the point of view that sees continuations as objects (usually subroutines) representing "the rest" or "the future" of a computation. The two describe the very same thing: a suspended continuation, is an object that, when resumed or "invoked", carries out the rest of some computation.

A delimited continuation is a sequential sub-program with an entry point (like a thread), which we'll call simply the entry point (in Scheme, this is the reset point), which may suspend or yield execution at some point, which we'll call the suspension point or the yield point (the shift point in Scheme). When a delimited continuation suspends, control is passed outside of the continuation, and when it is resumed, control returns to the last yield point, with the execution context up to the entry point intact. There are many ways to present delimited continuations, but to Java programmers, the following rough pseudocode would explain it best:

```
foo() { // (2)
  ...
  bar()
  ...
}

bar() {
  ...
  suspend // (3)
  ... // (5)
}

main() {
  c = continuation(foo) // (0)
  c.continue() // (1)
  c.continue() // (4)
}
```

A continuation is created (0), whose entry point is foo; it is then invoked (1) which passes control to the entry point of the continuation (2), which then executes until the next suspension point (3) inside the bar subroutine, at which point the invocation (1) returns. When the continuation is invoked again (4), control returns to the line following the yield point (5).

A rough outline of a possible API is presented below:

```
class _Continuation {
    public _Continuation(_Scope scope, Runnable target)
    public boolean run()
    public static _Continuation suspend(_Scope scope,Consumer<_Continuation> ccc)

    public ? getStackTrace()
}
```

Project Loom: Changes to the JVM

Dette er jeg veldig spent på. Jeg har noen ideer, skal skissere dem her:

Jeg tror at vi vil få en ny 'new' instruksjon som genererer en class instance med en egen stack. La oss kalle den 'new continuation'.

I tillegg til å ha sin egen stack, håper jeg slike instancer vil få muligheten til å lagre et 'reactivation point' (in Loom: suspension point).

I JVM-spesifikasjonen står det idag(versjon 9):

2.5.1 The `pc` Register

The Java Virtual Machine can support many threads of execution at once (JLS §17). Each Java Virtual Machine thread has its own `pc` (program counter) register. At any point, each Java Virtual Machine thread is executing the code of a single method, namely the current method (§2.6) for that thread. If that method is not `native`, the `pc` register contains the address of the Java Virtual Machine instruction currently being executed. If the method currently being executed by the thread is `native`, the value of the Java Virtual Machine's `pc` register is undefined. The Java Virtual Machine's `pc` register is wide enough to hold a `returnAddress` or a native pointer on the specific platform.

En Class Instance som reprsenterer hodet på en 'Continuation' kaller vi en Continuation Instance.

Vi lar Continuation Instance få to nye felt:

1) ENC – Enclosure, dvs. En pointer til omsluttende Continuation Instance eller en tråd.
2) PC -- Program Counter. Vi kan bruke det som 'Program Part of a Reactivation Point'.

Example Detach – **Call** – Detach

Denne gangen med separat Stack
Knyttet til Componentene

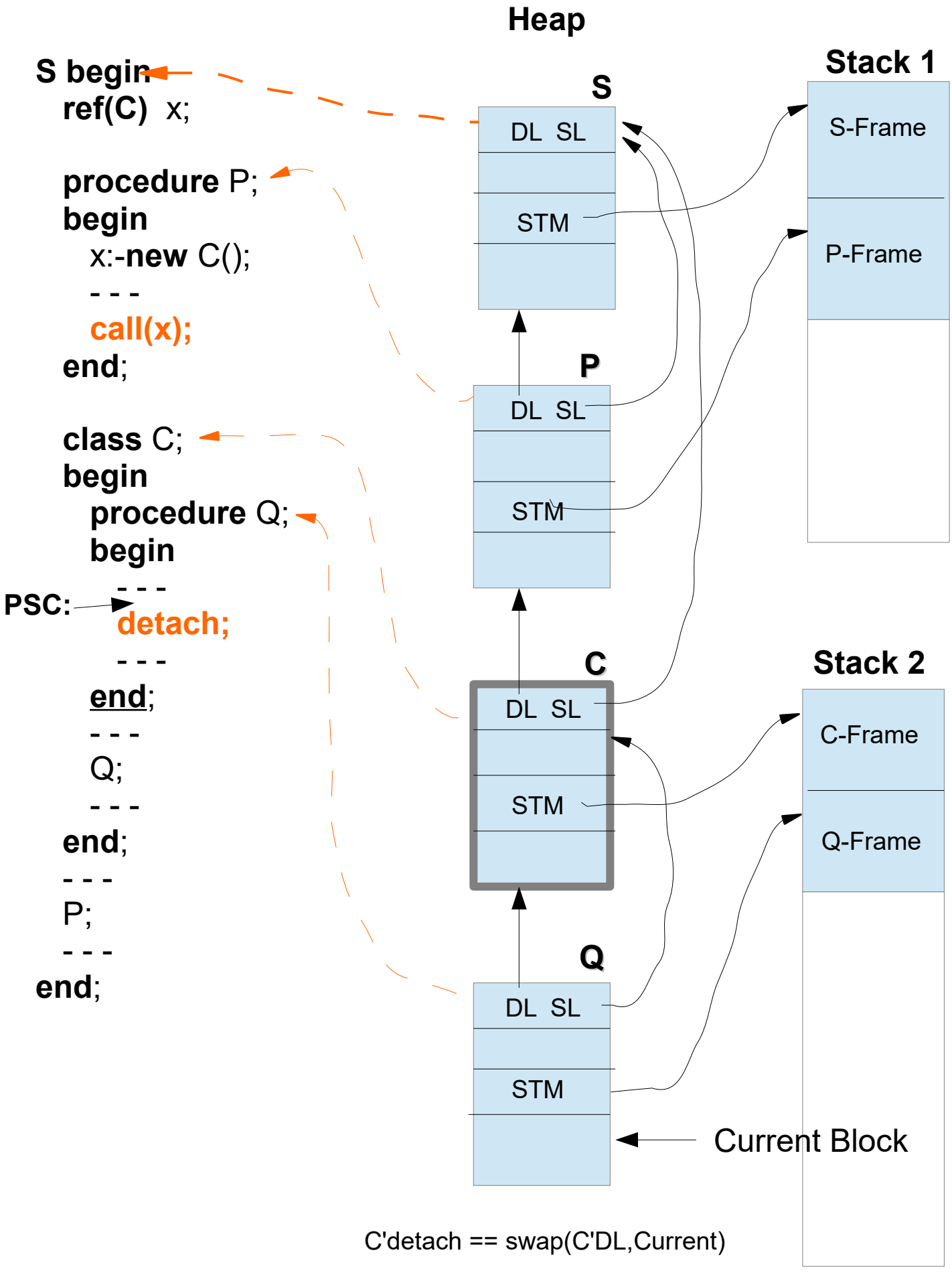

Fig. 5 Separat Stack: Just before Detach

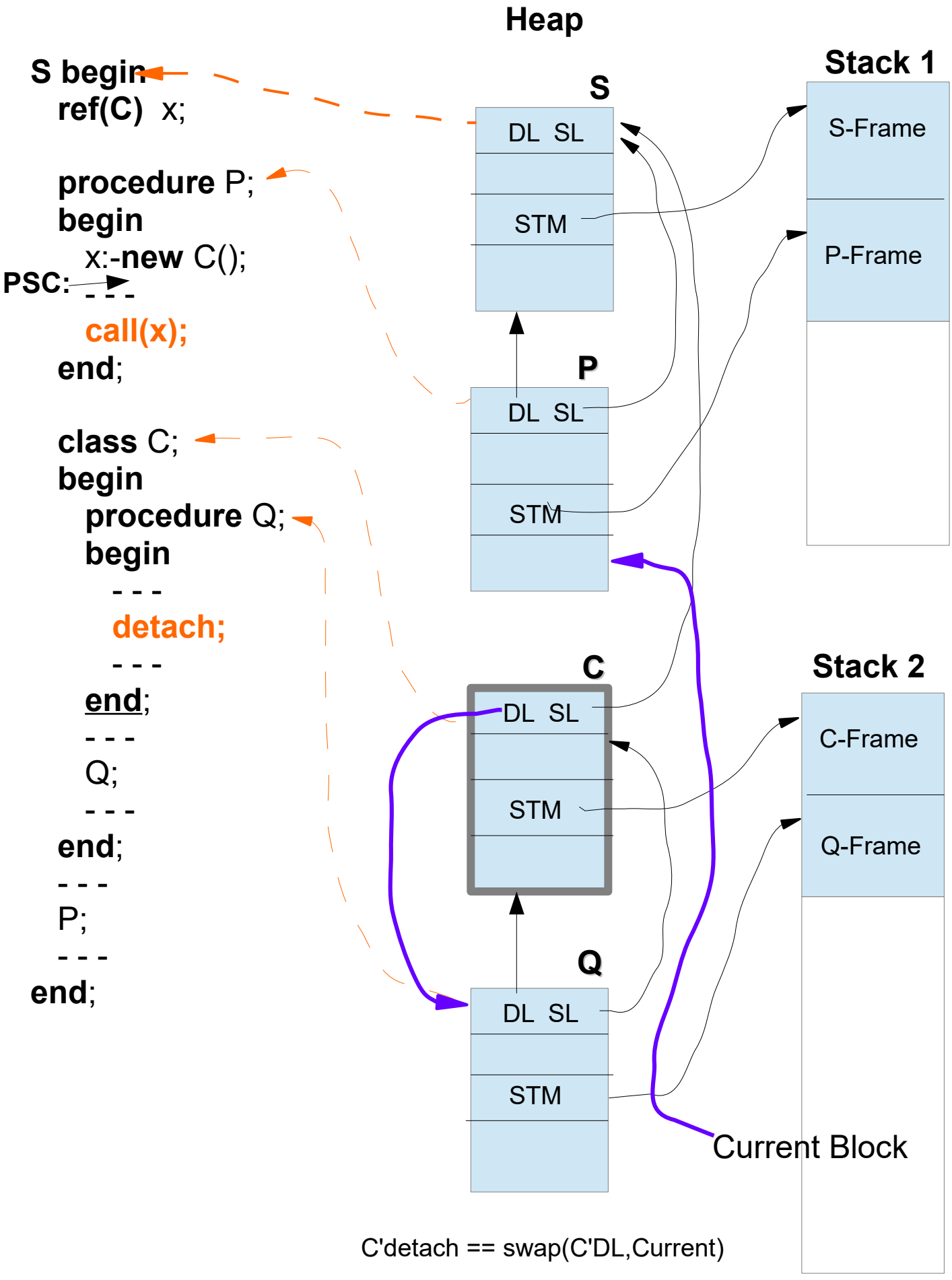

Fig. 6 Just after Detach

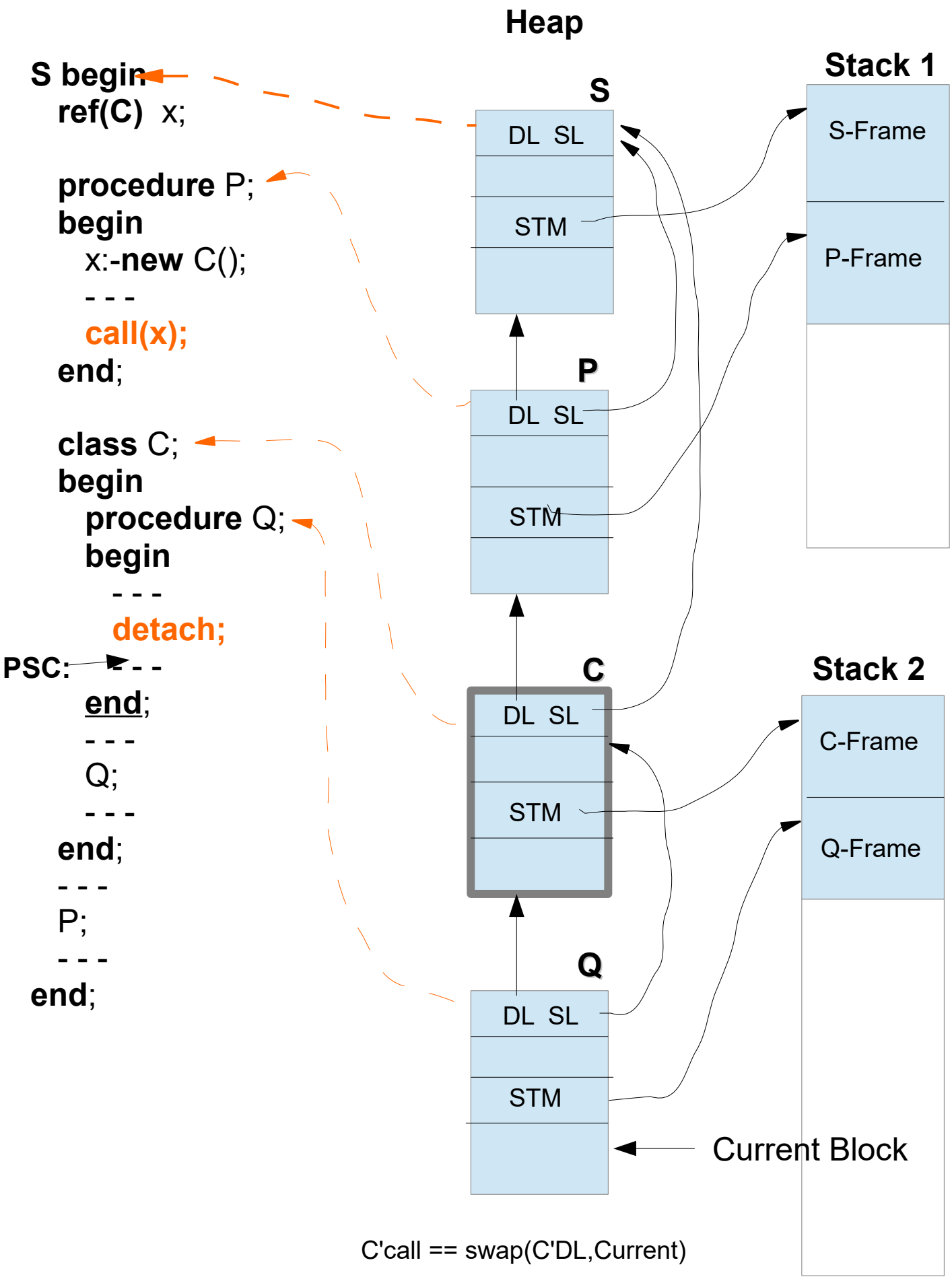

Fig. 7 Just after Call

Example Detach – **Resume** – Detach

Med separate Stack'er

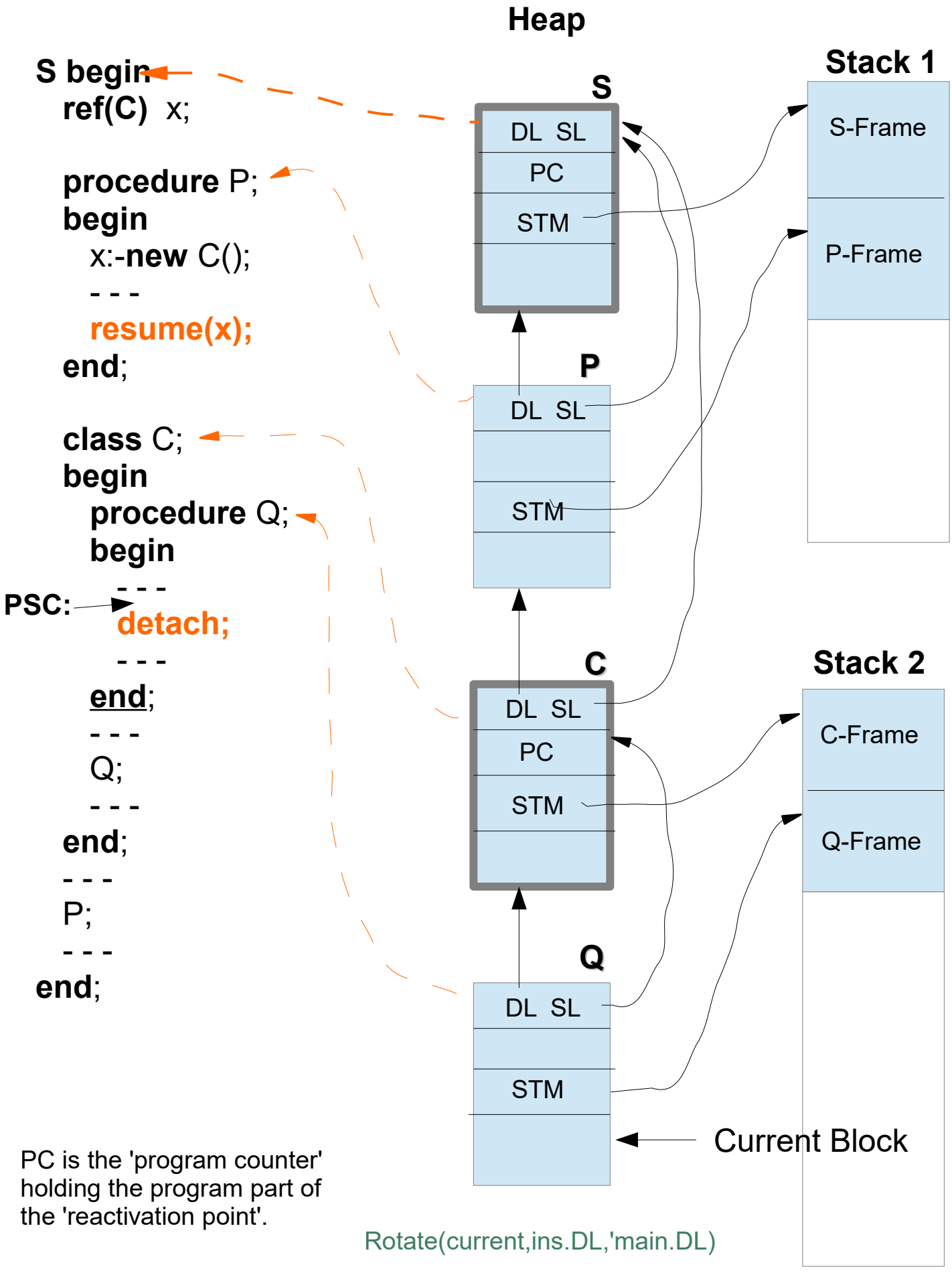

Fig. 8 Separat Stack: Just before 1.Detach

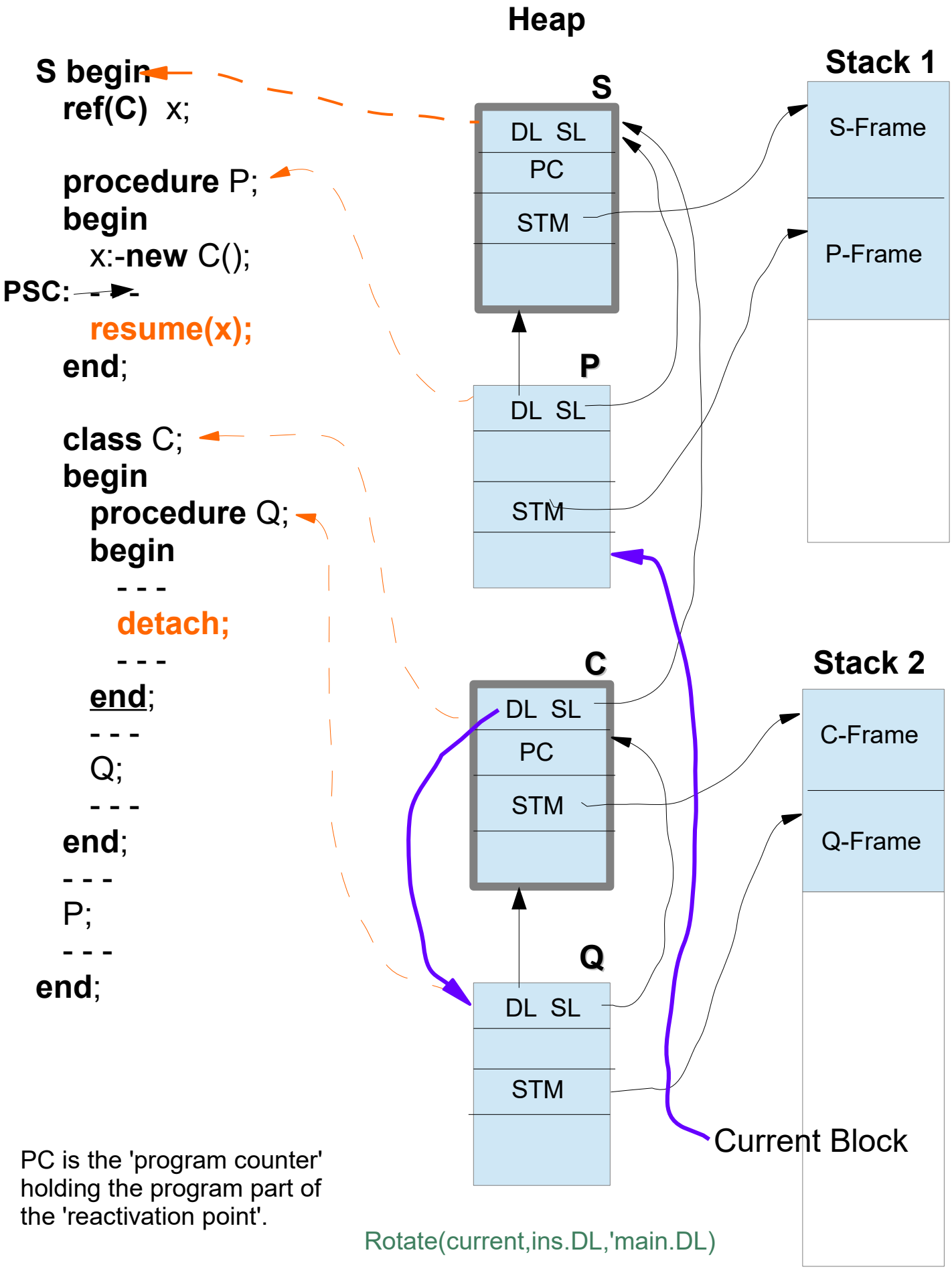

Fig. 9 Just after 1.Detach and before Resume

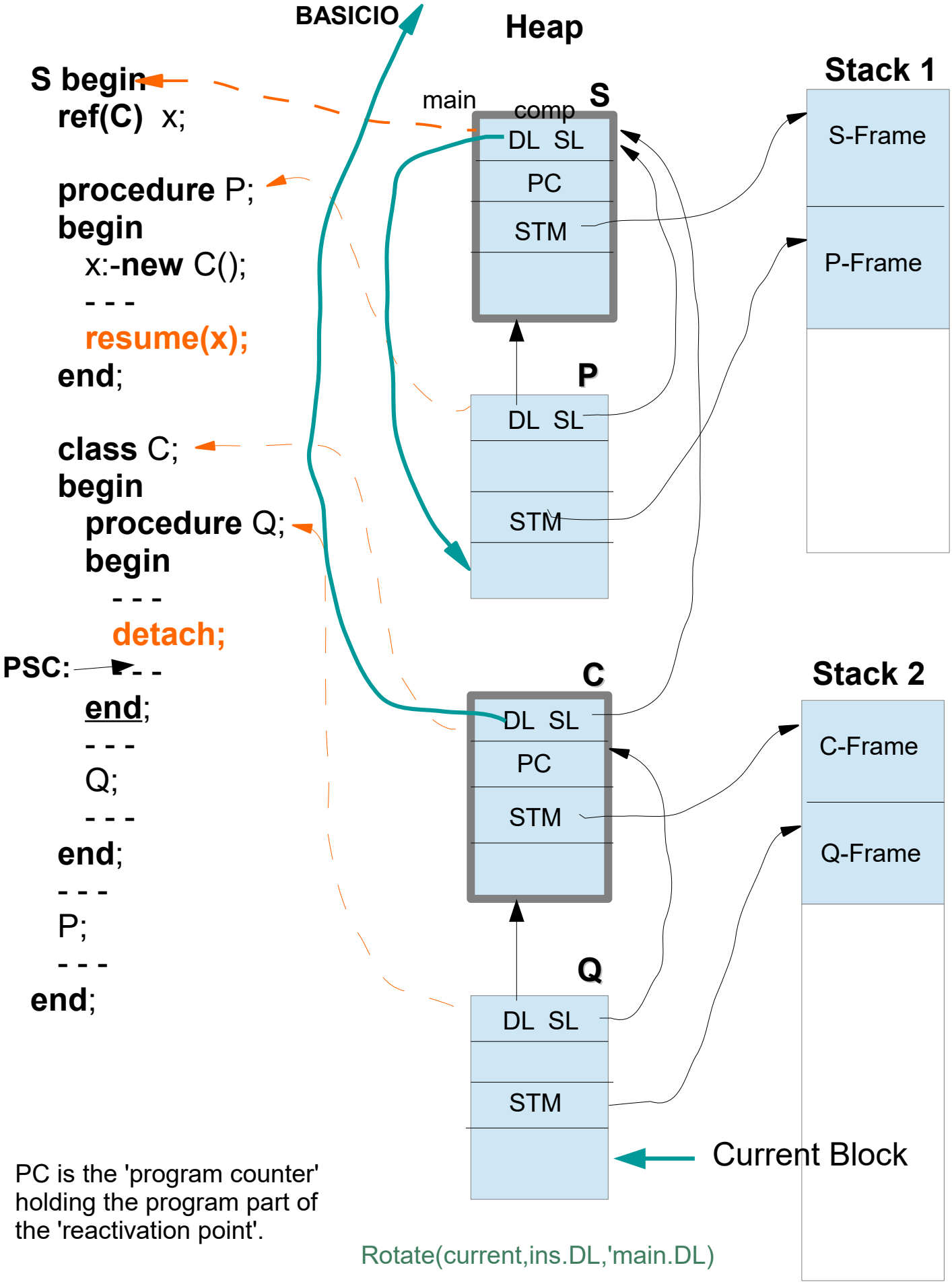

Fig. 10 Just after Resume

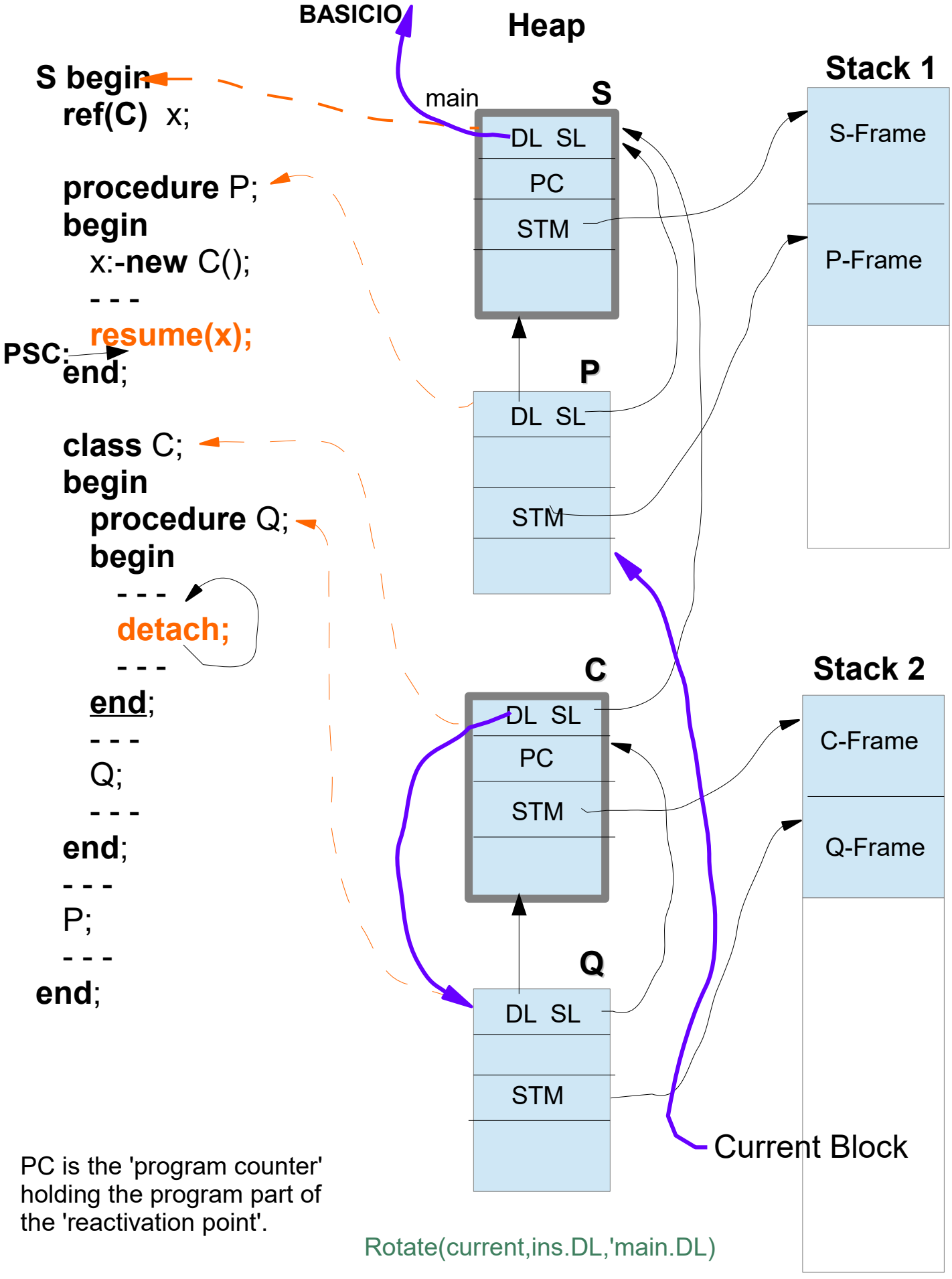

Fig. 11 Just after 2. Detach

Jøss - Er det så enkelt ?

I så fall må bruker vi Thread's inntil videre

De klassene som er markert med tykk ramme gjøres om til Tråder

Altså subklasser av QPObject istenfor BASICIO:

```
class QPObject extends BASICIO implements Runnable {
   Thread thread;
}
```

Constructor'n til den genererte klassen endres til:

```
public CoroutineSample(RTObject staticLink) {
   super(staticLink);
   BPRG(blockLevel);
   // Parameter assignment to locals
   // Start Object in a new Thread
   thread=new Thread(this);
   thread.start(); Thread.currentThread().suspend();
}
```

Vi starter altså opp en ny tråd samtidig som vi suspender current.

Terminering, f.eks ved at vi når final-end i klassen, eller ved en non-local goto utføres ved kall på RTS-metoder:

- ECB End Class Body (beskrevet på en av de følgende sider)
- $LABQNT'GOTO (denne beskriver jeg ikke nå)

Selve avslutningen av tråden gjøres ved å 'gi den til GarbageCollector'n.

```
current= ...
if(this instanceof QPObject){
   ((QPObject)current).thread.resume();
   ((QPObject)this).thread=null; // Leave it to GC
}
```

```
public void DETACH()
{ RTObject ins;   // Instance to be detached.
  RTObject dl;    // Temporary reference to dynamical enclosure
  RTObject main;  // The head of the main component and also
                  // the head of the quasi-parallel system.
  ins=this;
  //  Detach on a prefixed block is a no-operation.
  if(!current.isQPSystemBlock())
  { // Make sure that the object is on the operating chain.
    // Note that a detached or terminated object cannot be on
    // the operating chain.
    dl=current;
    while(dl!=ins) {
        dl=dl.DL;
        if(dl==null) throw
          new RuntimeException("not on operating chain.");
    }

    // Treat the case resumed and operating first,
    // because it is probably the dynamically most common.
    if(ins.state==OperationalState.resumed) {
        ins.state=OperationalState.detached;
        // Find main component for component to be detached.
        // The main component head must be the static enclosure
        // of the object, since the object has been RESUMEd.
        main=ins.SL;
        // Ignore any temporary(aStack) objects.
        while(main.DL.$LSC==null) main=main.DL;
        while(ins.DL.$LSC==null) ins=ins.DL;
        // Rotate:
        // <main.DL,ins.DL,current>:=<ins.DL,current,main.DL>
        dl=main.DL; main.DL=ins.DL; ins.DL=current;
        current=dl;
    } else {
        // Invariant: ins.state==OperationalState.attached
        ins.state=OperationalState.detached;
        // Ignore any temporary(aStack) objects.
        while(ins.DL.$LSC==null) ins=ins.DL;
        // Swap the contents of object's 'DL' and 'current'.
        // <ins.DL,current>:=<current,ins.DL>
        dl=ins.DL; ins.DL=current; current=dl;
    }
    // Invariant: Thread.currentThread()==this.thread
    ((QPObject)current).thread.resume();
    this.thread.suspend();
  }
}
```

```
public void CALL(RTObject ins)
{ RTObject dl; //  Temporary reference to dynamic enclosure.
  if(ins==null) throw new RuntimeException("x is none.");
  if(ins.state!=OperationalState.detached) throw
        new RuntimeException("x is not in detached state.");
  // The object to be attached cannot be on the operating
  // chain, because then its state would have been resumed
  // and not detached.

  // Swap the contents of 'current' and object's 'dl'.
  //  <ins.DL,current>:=<current,ins.DL>;
  dl=ins.DL; ins.DL=current; current=dl;
  // From now on the object is in attached state.
  // It is no longer a component head.
  ins.state=OperationalState.attached;
  //updateContextVector(); // TODO: ???
  ((QPObject)current).thread.resume(); this.thread.suspend();
}
```

```
public void RESUME(RTObject ins)
{ RTObject comp;   //  Component head.
  RTObject mainSL; //  Static enclosure of main component head.
  RTObject main;   //  The head of the main component and also
                   //  the head of the quasi-parallel system.
  if(ins==null)
      throw new RuntimeException("Resume(x): x is none.");

  if(ins.state!=OperationalState.resumed) // A no-operation?
  { // The object to be resumed must be local to a system head.
    main=ins.SL;
    if(!main.isQPSystemBlock()) throw
         new RuntimeException("x is not QPS-component.");
    if(ins.state!=OperationalState.detached) throw
         new RuntimeException("x is not in detached state.");
    // Find the operating component of the QPS system.
    comp=current; mainSL=main.SL;
    while(comp.DL!=mainSL) comp=comp.DL;
    if(comp.state==OperationalState.resumed)
        comp.state=OperationalState.detached;
    // Rotate the contents of 'ins.dl','comp.dl' and 'current'.
    // Invariant:   comp.DL = mainSL
    //  <ins.DL,comp.DL,current>=<comp.DL,current,ins.DL>
    comp.DL=current; current=ins.DL; ins.DL=mainSL;
    ins.state=OperationalState.resumed;
    //updateContextVector(); // TODO: ???
    ((QPObject)current).thread.resume(); this.thread.suspend();
  }
}
```

```
public void ECB()
  { RTObject dl;   // A temporary to the instance dynamically
                   // enclosing the resumed object ('current').
    RTObject main; // The head of the main component and also
                   // the head of the quasi-parallel system.
    if(current.state==OperationalState.attached) {
    current.state=OperationalState.terminated;
        // Make the dynamic enclosure the new current instance.
        current=current.DL;
    } else if(current.state==OperationalState.resumed) {
        //  Invariant:  current.DL = main.SL
        current.state=OperationalState.terminated;
        // Find main component (and system) head.
        // It must be the static enclosure since the object
        // has been RESUMEd.
        main=current.SL;
        // The main component becomes the operating component.
        dl=current.DL; current.DL=null;
        current=main.DL; main.DL=dl;
    } else {
        //  Treat the case of a prefixed block instance.
        current=current.DL;
    }
    if(this instanceof QPObject)
    { ((QPObject)current).thread.resume();
      ((QPObject)this).thread=null; // Leave it to the GC
    }
  }
```