Simula QPS Implemented using 'Delimited Continuation'

(as of 15 mars 2019)

Slik var det på 60'tallet:

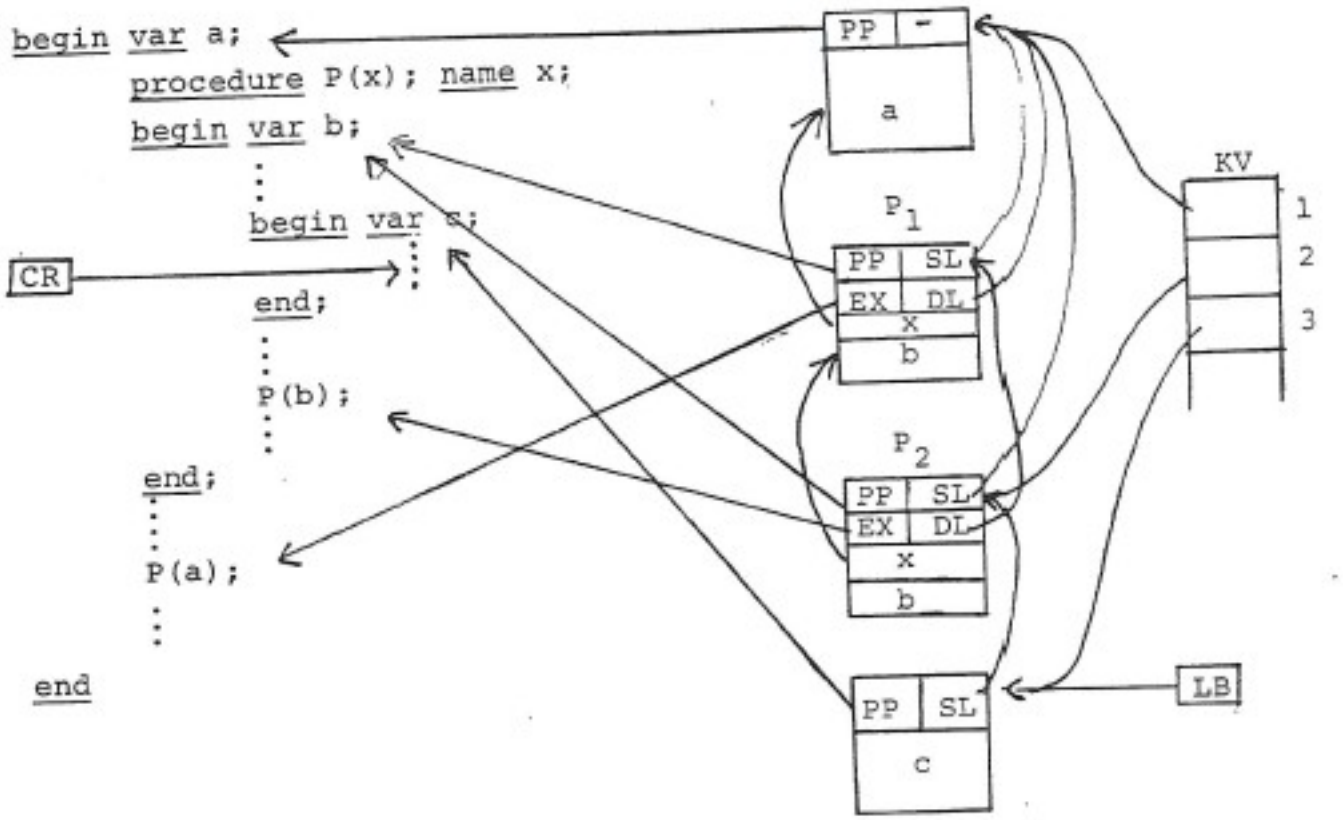


Fig. 5 Skjematisk program-eksempel.

Copied from " RUNTIME ORGANISASJON FOR ALGOL / SIMULA " av Ole-Johan Dahl.

LOOM: Delimited continuations

```
public class Continuation implements Runnable {
    public Continuation(ContinuationScope scope, Runnable target)
    public final void run()
    public static void yield(ContinuationScope scope)
    public boolean isDone()
        ....
}
```

A continuation object is constructed by passing two arguments to the constructor:
- - a Runnable target that serves as the body of the continuation, and a java.lang.ContinuationScope. The scope is the delimited continuation's prompt, that allows continuations to be nested. One could think of such "scoped continuations" as nested try/catch blocks, where the scope is the type of the exception thrown, which determines the handler called.
A continuation is started by calling Continuation.run, which would start executing the body in the continuation's target, and returns either when the continuation terminates (the body runs to completion, and terminates either normally or abnormally), or when it yields on the continuation's scope. To query the reason for run returning, use Continuation.isDone, which returns true if the body has terminated, or false if it has yielded.
A call to the static Continuation.yield suspends the current continuation and all enclosing continuations up until the innermost one with the scope passd to yield, causing the run method of that continuation to return.
The Continuation class is implemented natively in Hotspot (except for scoping; that is implemented in Java). Every continuation has its own stack. From the perspective of the implementation, starting or continuing a continuation mounts it and its stack on the current thread – conceptually concatenating the continuation's stack to the thread's – while yielding a continuation unmounts or dismounts it.

Example: Co-routine with three active phases:

```
static final ContinuationScope scope=new ContinuationScope("TST");

Continuation coroutine=new Continuation(scope,new Runnable() {
      public void run() {
            System.out.println("Part 1 - Statements");
            Continuation.yield(scope); // DETACH 1
            System.out.println("Part 2 - Statements");
            Continuation.yield(scope); // DETACH 2
            System.out.println("Part 3 - Statements");
      }});
```

An application may look like this:

```
coroutine.run(); // Will perform Part 1.
System.out.println("Returns here after first DETACH(Yield)");
coroutine.run(); // Will perform Part 2.
System.out.println("Returns here after second DETACH(Yield)");
coroutine.run(); // Will perform Part 3.
System.out.println("Returns here after 'FINAL END'");
coroutine.run(); // IllegalStateException: Continuation terminated
```

We get the following output:

```
Part 1 - Statements
Returns here after first DETACH(Yield)
Part 2 - Statements
Returns here after second DETACH(Yield)
Part 3 - Statements
Returns here after 'FINAL END'
Exception: IllegalStateException: Continuation terminated
```

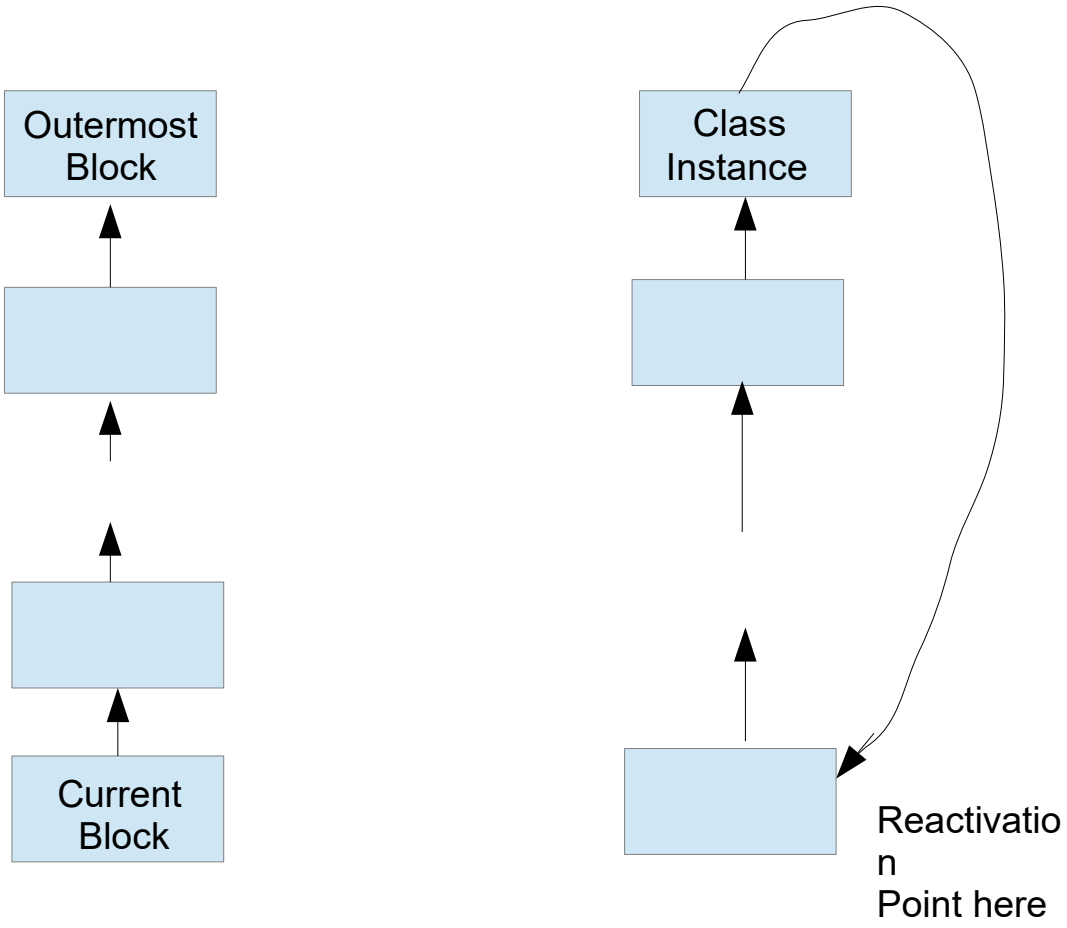
Example: Chain Execution (ala' Simulation)

```
static Continuation next;
static Continuation producer=null;
static Continuation consumer=null;

public static void EXECUTE(Continuation cont) {
      next=cont;
      while(next!=null) {
            Continuation n=next; next=null; n.run();
      }
}

public static void RESUME(Continuation cont) {
      next=cont; Continuation.yield(scope);
}

public static void example2() {

      producer=new Continuation(scope,new Runnable() {
            public void run() {
               System.out.println("Producer:Part 1 - Statements");
               RESUME(consumer);
               System.out.println("Producer:Part 2 - Statements");
               RESUME(consumer);
               System.out.println("Producer:Part 3 - Statements");
               RESUME(consumer);
      }});

      consumer=new Continuation(scope,new Runnable() {
            public void run() {
               System.out.println("Consumer:Part 1 - Statements");
               RESUME(producer);
               System.out.println("Consumer:Part 2 - Statements");
               RESUME(producer);
               System.out.println("Consumer:Part 3 - Statements");
      }});

      EXECUTE(producer); // Will start Producer:Part 1.
      System.out.println("End of program");
}
```

We get the following output:

```
Producer:Part 1 - Statements
Consumer:Part 1 - Statements
Producer:Part 2 - Statements
Consumer:Part 2 - Statements
Producer:Part 3 - Statements
Consumer:Part 3 - Statements
End of program
```

Klassisk illustrering av Detach



En liten avsporing:

Hva handler 'Invariant-2' feilen om ?

Problemet er at runtime designet er basert på invarianten:

Det er aldri mer enn en Thread som kjører.

Hvis vi kan stole på dette så holder det å ha en statisk variabel 'CUR$' som peker på 'current simula object'.

Følgende kode-sekvens illustrerer problemet:

```
protected static void SWAP_THREAD(final Thread next) {
    Thread prev=Thread.currentThread();
    next.notify(); // resume next
    prev.wait();   // suspend prev
    // KOMMER HIT NÅR 'prev' VEKKES IGJEN
    waitUntilPrevThreadNotRunning();
}
```

Vi ønsker at 'current Thread' skal suspenderes og 'next Thread' starte opp i en og samme operasjon.

For at det skal skje, må 'next Thread' vente litt slik at 'current Thread' rekker å bli suspendert.

Jeg tror at problemet er løst ved at metoden waitUntilPrevThreadNotRunning er innført.

Denne bruker en uoffisiell test på om 'prevThread' er 'running' og venter på at det endrer seg.

```
while(prevThread.getState()==Thread.State.RUNNABLE) wait
```

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.

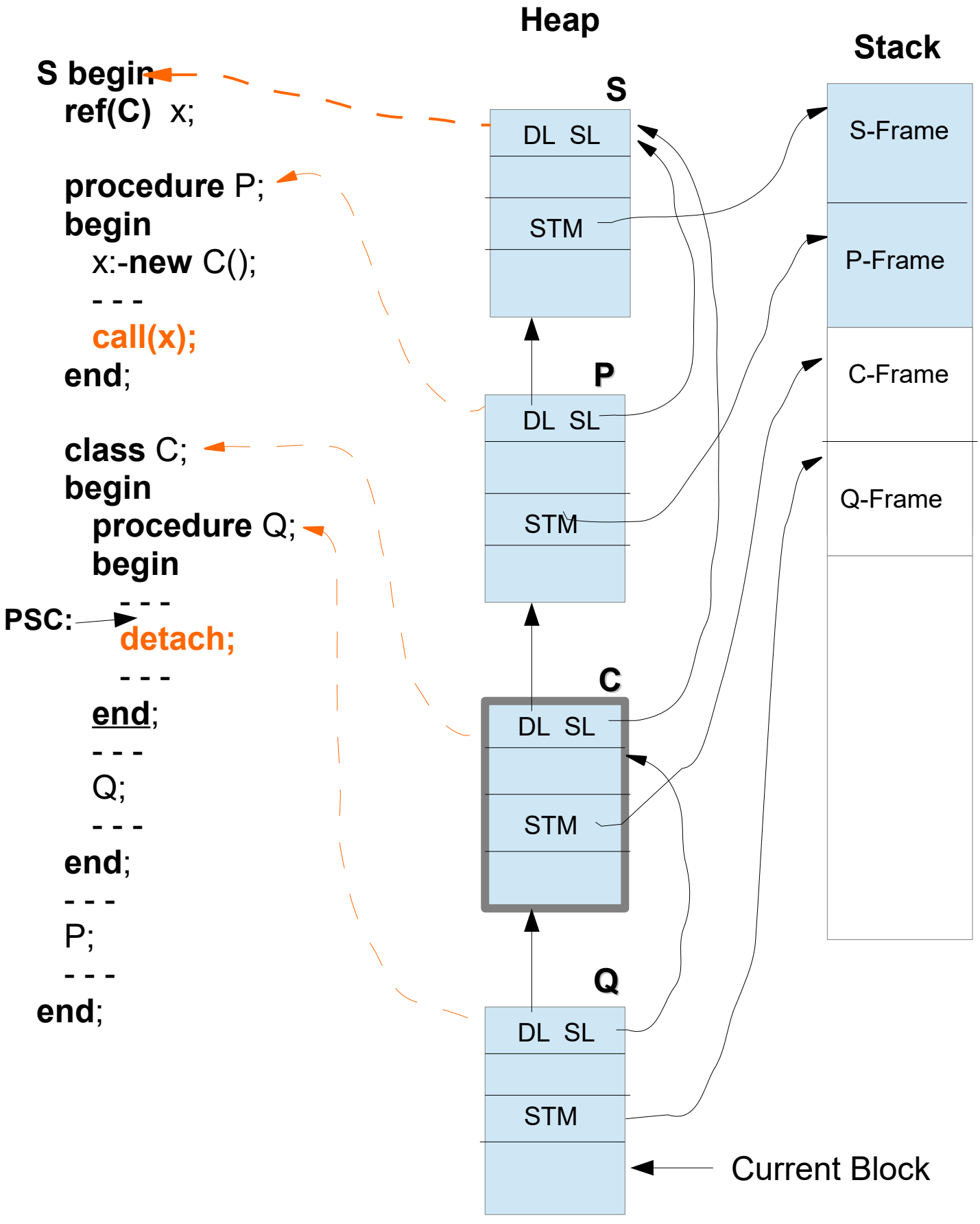
Legg merke til at Stack-Framen til Q blir frigjort mens framen til C blir hengende og sansynligvis bryter reglene for bruk av stacken.

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.

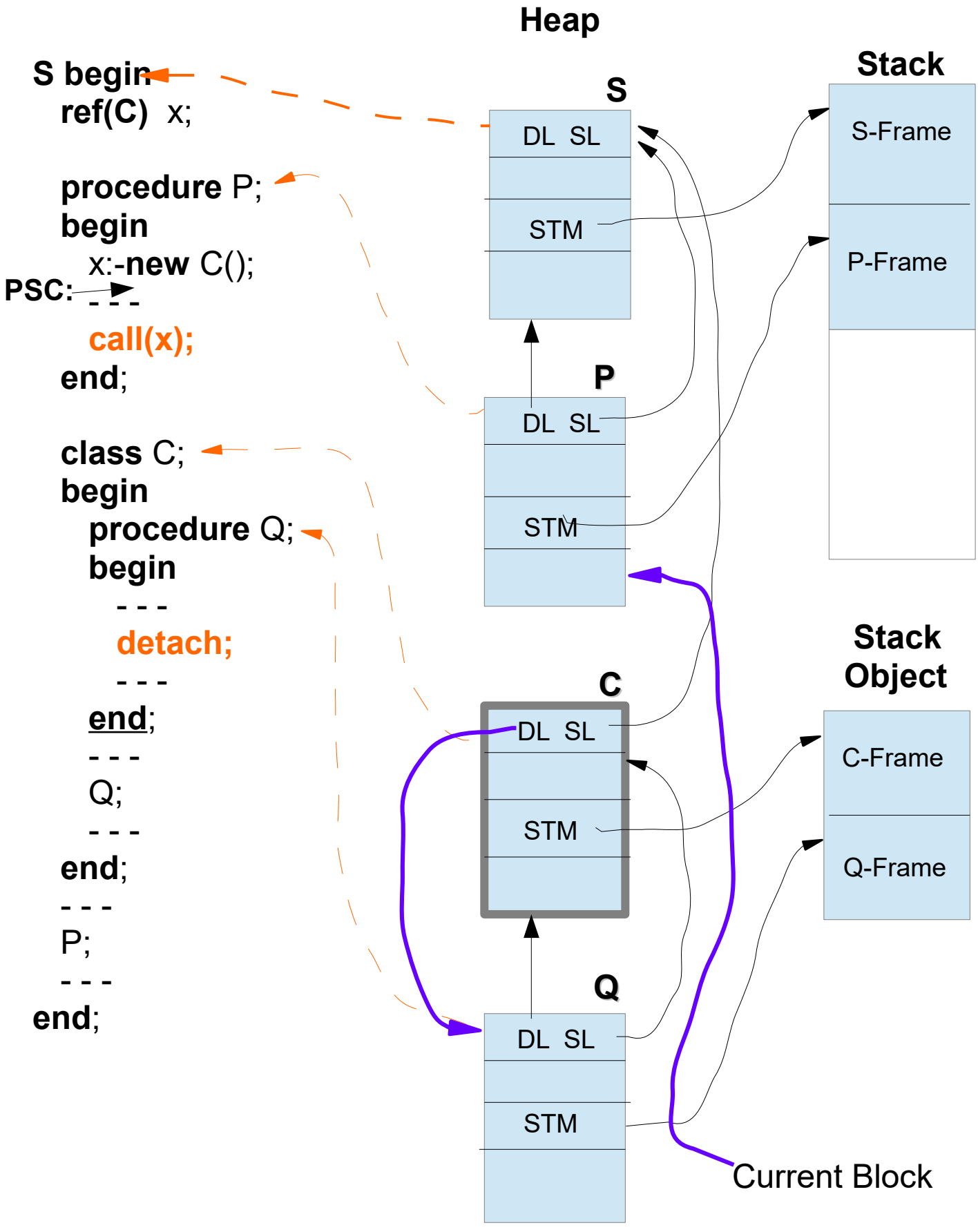
Example Detach – **Call** – Detach

Denne gangen med separat Stack
Knyttet til Componentene



C'detach == swap(C'DL,Current)

Fig. 5 Using LOOM: Just before Detach



C'detach == swap(C'DL,Current)

Fig. 6 Just after Detach – Dismounts part of Stack

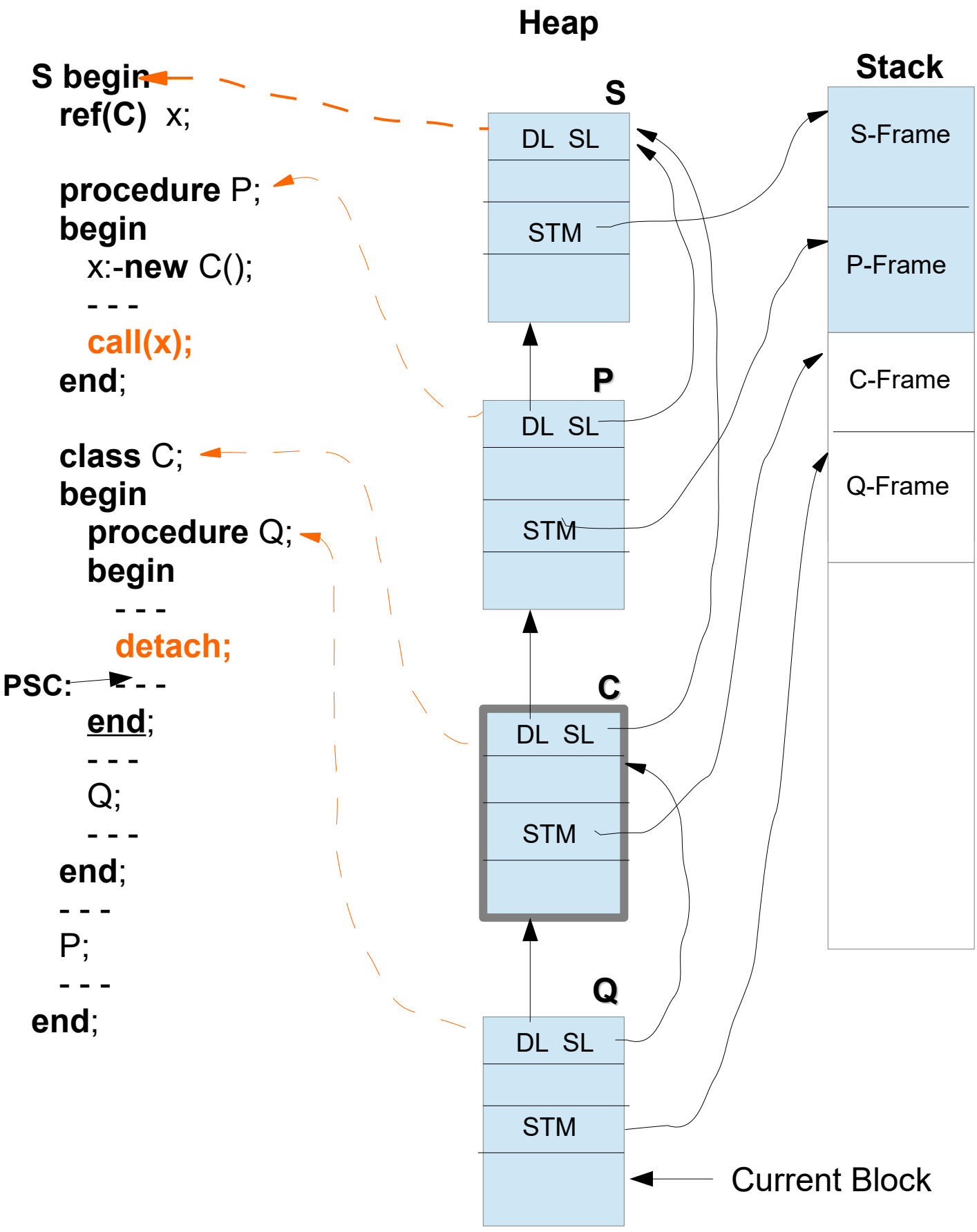


Fig. 7 Just after Call – Stack is re-mounted

Example Detach – **Resume** – Detach

Med separate Stack'er

MERK: Resten av dokumentet

ER IKKE OPPDATERT !

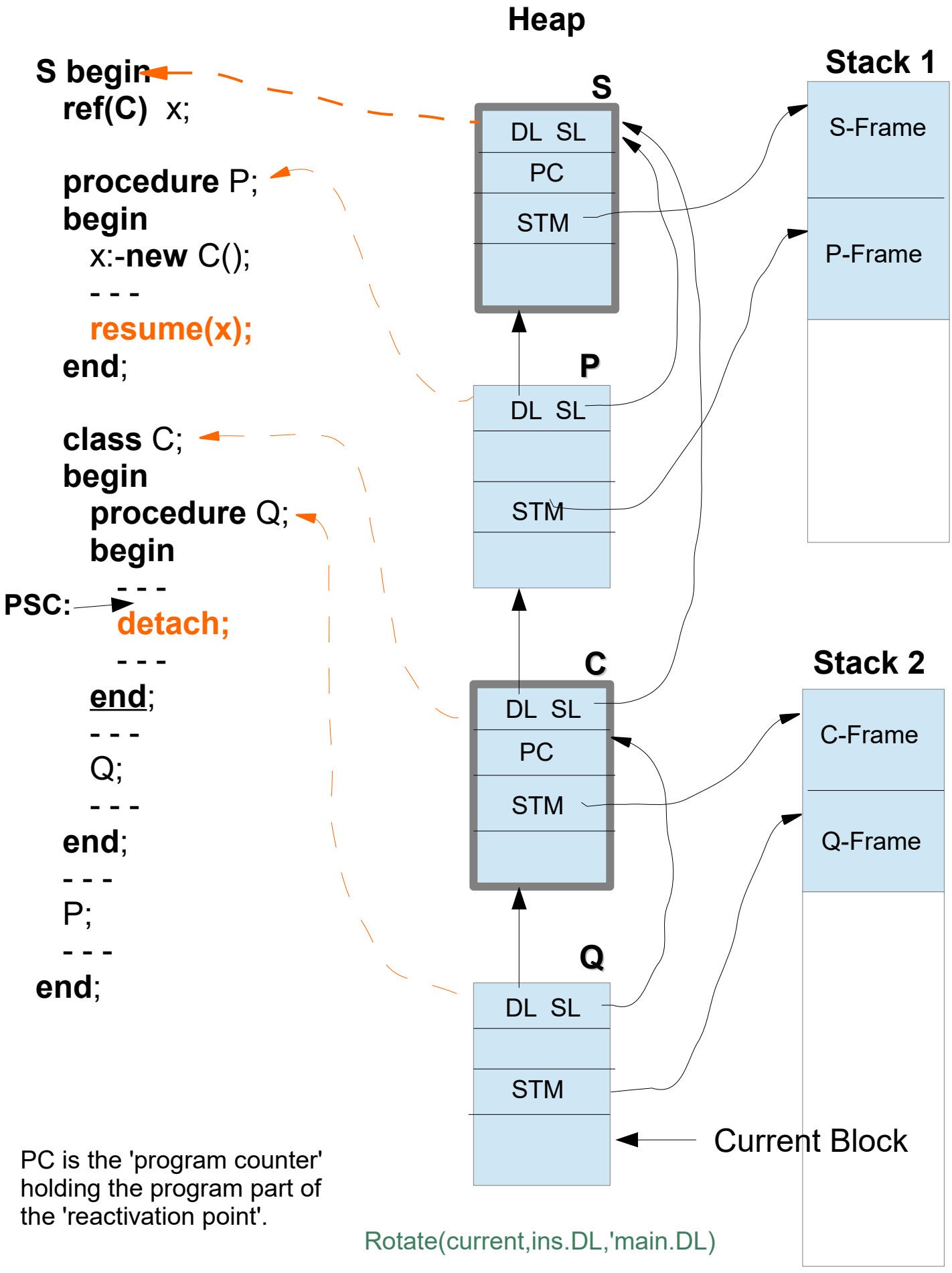


Fig. 8 Separat Stack: Just before 1.Detach

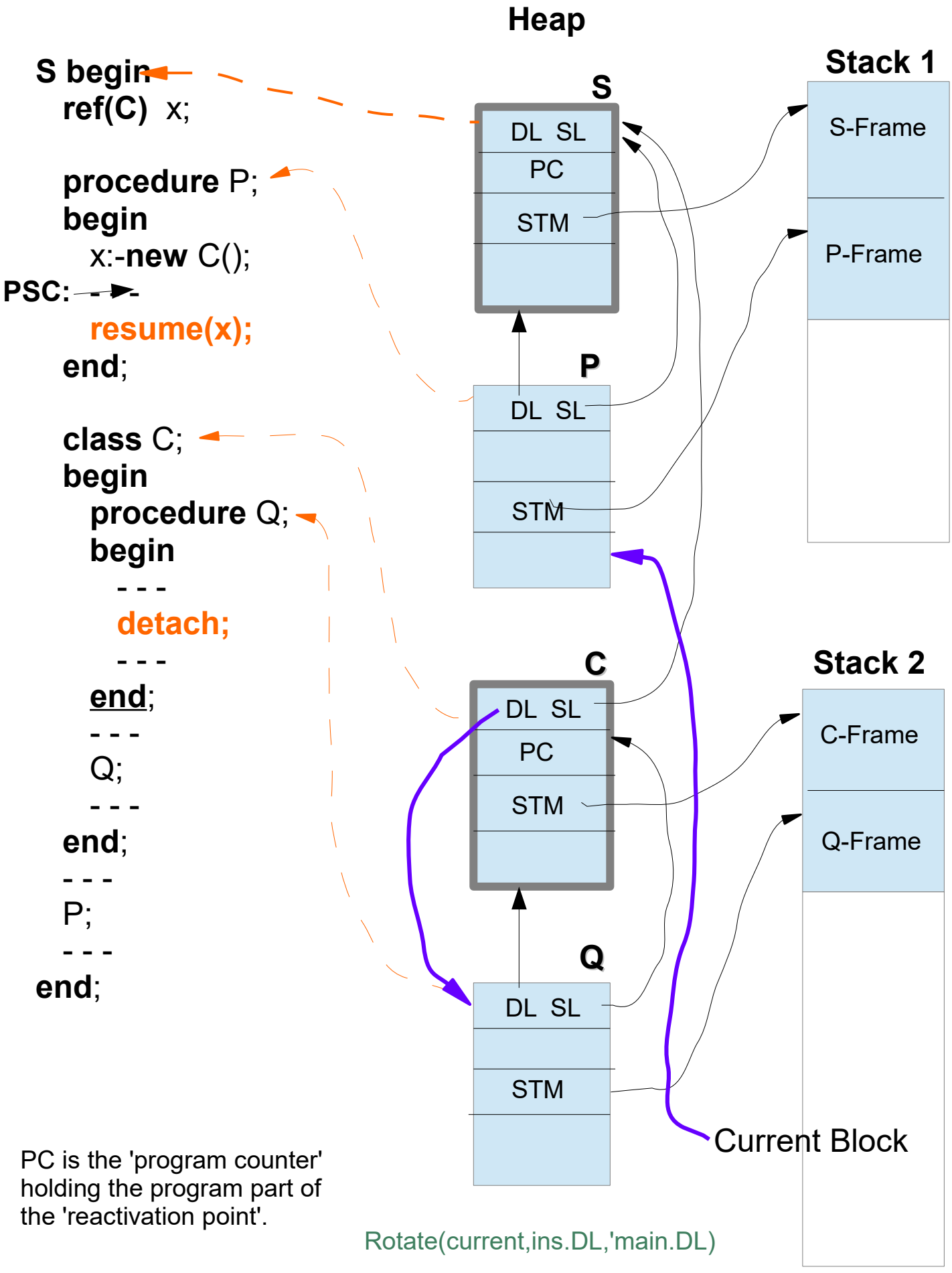


Fig. 9 Just after 1.Detach and before Resume

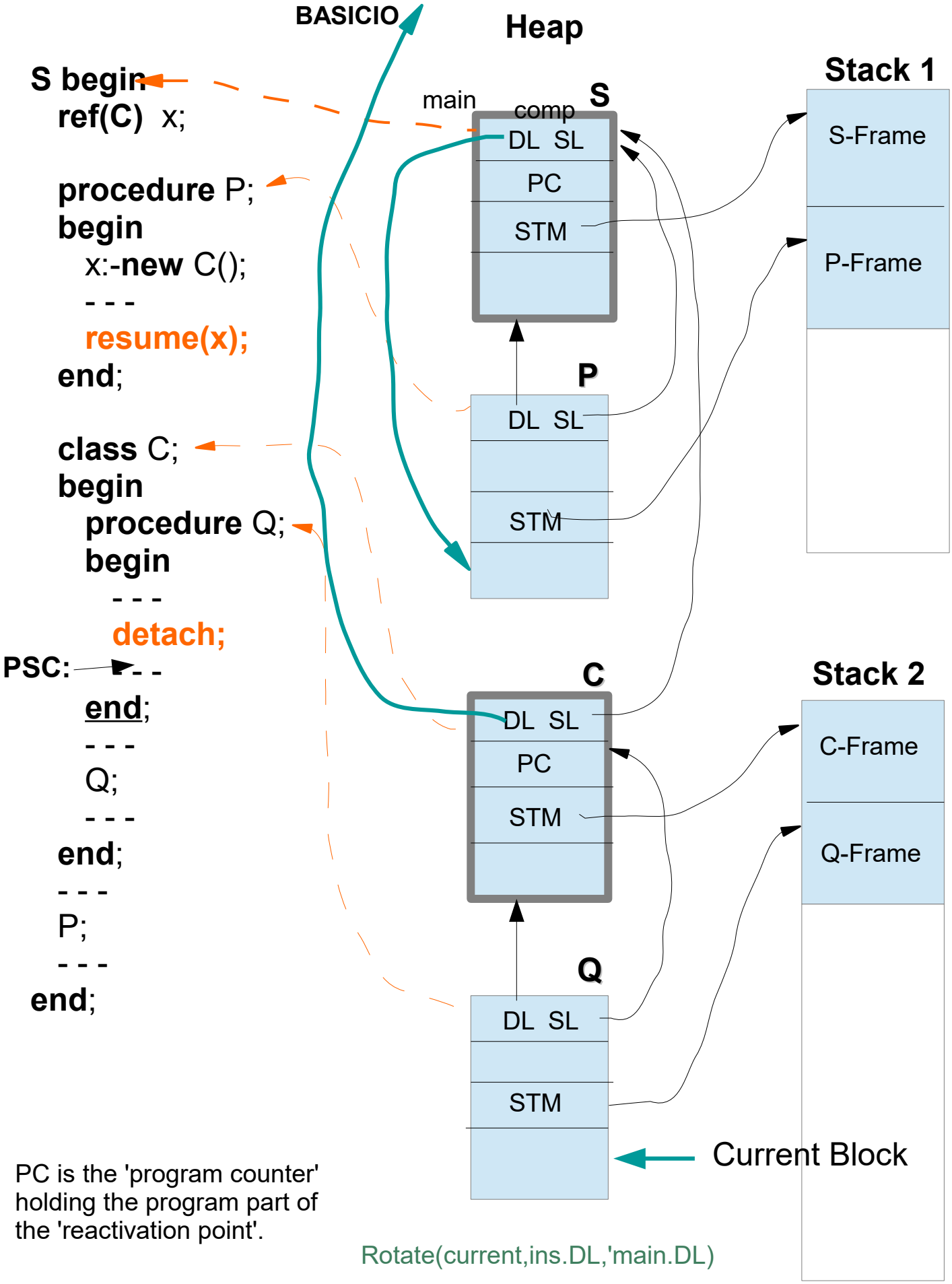


Fig. 10 Just after Resume

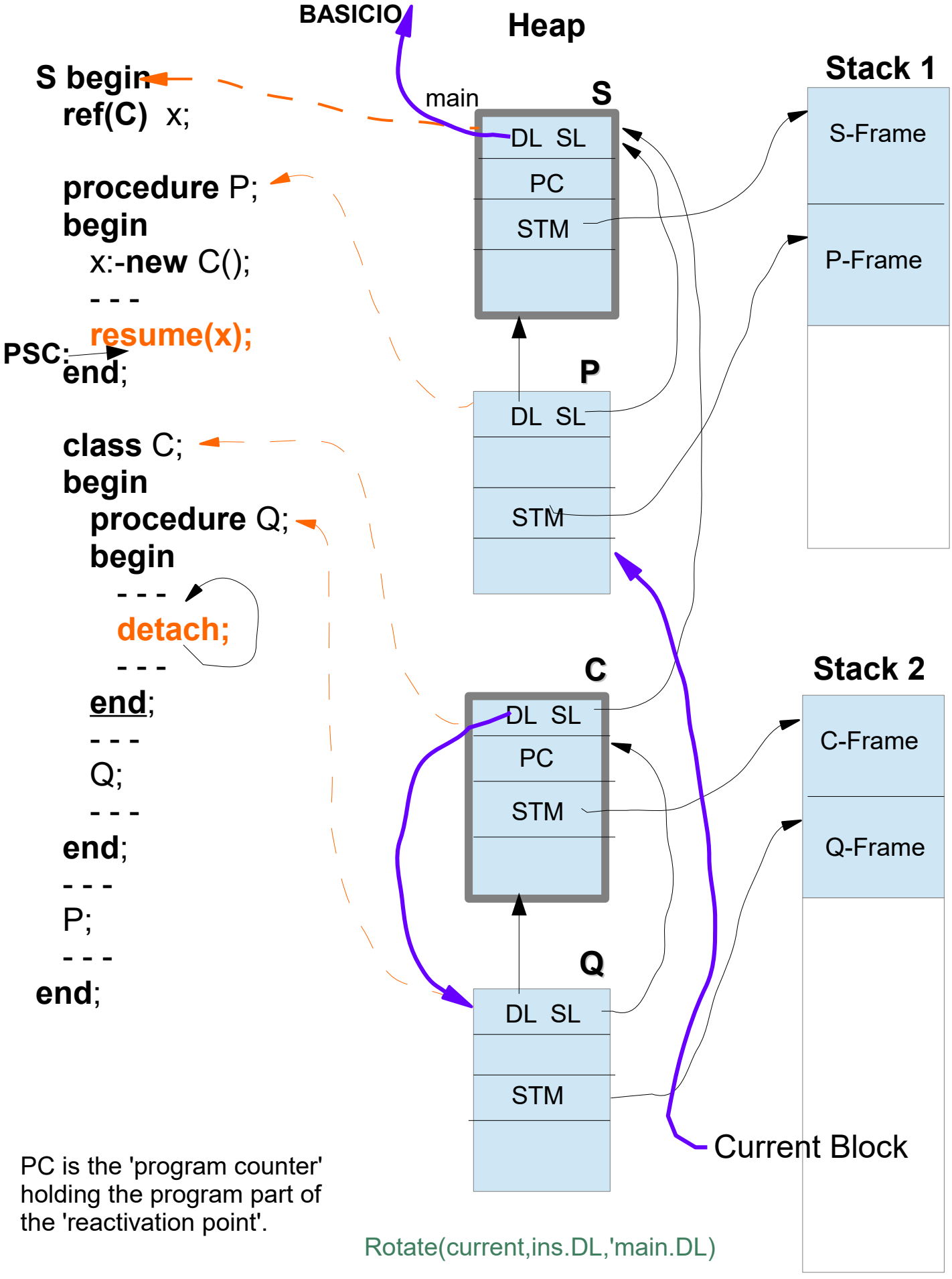


Fig. 11 Just after 2. Detach

```
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(!CUR$.isQPSystemBlock())
  { // Make sure that the object is on the operating chain.
       // Note that a detached or terminated object cannot be on
       // the operating chain.
       dl=CUR$;
       while(dl!=ins) {
           dl=dl.DL$;
           if(dl==null) throw
             new RuntimeException("x.Detach: x is not on the operating chain.");
       }
       // Treat the case resumed and operating first,
       // because it is probably the dynamically most common.
       if(ins.STATE$==OperationalState.resumed) {
           ins.STATE$=OperationalState.detached;
           // Find main component for component to be detached. The main
           // component head must be the static enclosure of the object,
           // since the object has OperationalState.resumed.
           main=ins.SL$;
           // Rotate the contents of 'CUR$', 'ins.DL$' and 'main.DL$'.
           // <main.DL$,ins.DL$,CUR$>:=<ins.DL$,CUR$,main.DL$>
           dl=main.DL$; main.DL$=ins.DL$; ins.DL$=CUR$; CUR$=dl;
       } else {
           ins.STATE$=OperationalState.detached;
           // Swap the contents of object's 'DL$' and 'CUR$'.
           // <ins.DL$,CUR$>:=<CUR$,ins.DL$>
           dl=ins.DL$; ins.DL$=CUR$; CUR$=dl;
       }
       Util.ASSERT(CUR$.DL$!=CUR$,"Invariant");
       updateContextVector();
     swapThreads(ins.DL$);
  }
}
```

```
public void call(RTObject$ ins)
{ RTObject$ dl;     //  Temporary reference to dynamic enclosure.
  if(ins==null) throw new RuntimeException("Call(x): x is none.");
  if(ins.STATE$!=OperationalState.detached) throw
        new RuntimeException("Call(x): 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 'CUR$' and object's 'dl'.
  //  <ins.DL$,CUR$>:=<CUR$,ins.DL$>;
  dl=ins.DL$; ins.DL$=CUR$; CUR$=dl;
  // From now on the object is in attached state.
  // It is no longer a component head.
  ins.STATE$=OperationalState.attached;
  updateContextVector();
  swapThreads(ins.DL$);
}
```

```
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("Resume(x): x is not local"
                                +" to sub-block or prefixed block.");

    if(ins.STATE$!=OperationalState.detached) throw
         new RuntimeException("Resume(x): x is not in detached state"
                              +" but "+ins.STATE$);
    // Find the operating component of the quasi-parallel system.
    comp=CUR$; mainSL=main.SL$;
    while(comp.DL$!=mainSL)
    { Util.ASSERT(comp.DL$!=comp,"Invariant");
      comp=comp.DL$;
    }
    if(comp.STATE$==OperationalState.resumed)
             comp.STATE$=OperationalState.detached;
    // Rotate the contents of 'ins.dl', 'comp.dl' and 'CUR$'.
    // Invariant:   comp.DL$ = mainSL
    //  <ins.DL$,comp.DL$,CUR$>=<comp.DL$,CUR$,ins.DL$>
    comp.DL$=CUR$; CUR$=ins.DL$; ins.DL$=mainSL;
    ins.STATE$=OperationalState.resumed;
      updateContextVector();
    swapThreads(comp.DL$);
  }
}
```

```
public void EBLK()
  { RTObject$ dl;   // A temporary to the instance dynamically
                    // enclosing the resumed object ('CUR$').
    RTObject$ main; // The head of the main component and also
                    // the head of the quasi-parallel system.

    // Treat the attached case first, it is probably most common.
    if(CUR$.STATE$==OperationalState.attached) {
       CUR$.STATE$=OperationalState.terminated;
      //  Make the dynamic enclosure the new current instance.
      CUR$=CUR$.DL$;
    } else if(CUR$.STATE$==OperationalState.resumed) {
      // Treat the case of a resumed and operating object.
      //  It is the head of an object component. The class
      //  object enters the terminated state, and the object component
      //  disappears from its system. The main component of that system
      //  takes its place as the operating component of the system.
      //  Invariant:  CUR$.STATE$ = resumed  and  CUR$.DL = main.SL
      CUR$.STATE$=OperationalState.terminated;
      //  Find main component (and system) head. It must be the static
      //  enclosure since the object has been RESUMEd.
      main=CUR$.SL$;
      //  The main component becomes the operating component.
      dl=CUR$.DL$; CUR$.DL$=null;
      CUR$=main.DL$; main.DL$=dl;
    }
    if(CUR$==null || CUR$==CTX$) {
       SYSOUT$.outimage();
      // PROGRAM PASSES THROUGH FINAL END
      System.exit(0);
    } else {
       updateContextVector();
      if(this.THREAD$!=CUR$.THREAD$)
      { if(QPS_TRACING) TRACE("Resume "+CUR$.THREAD$);
        CUR$.THREAD$.resume();
        if(QPS_TRACING) TRACE("Terminate "+this.THREAD$);
        this.THREAD$=null; // Leave it to the GarbageCollector
      }
    }
  }
```