CSE 120: Principles of Operating Systems
Lecture 2: Processes

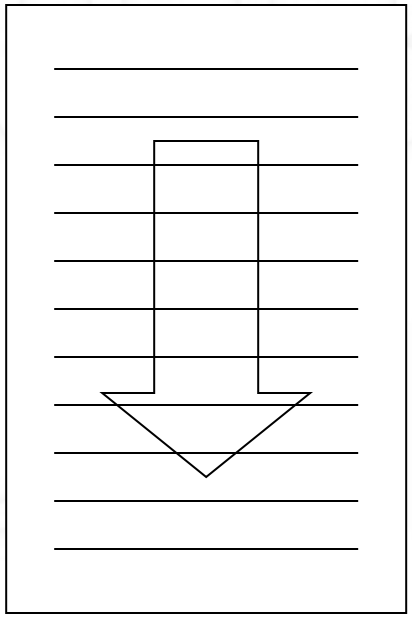
Prof. Joseph Pasquale
University of California, San Diego
January 11, 2023

Introduction

- Most basic kernel function: run a program
- Users wants ability to run multiple programs
- How to achieve given single CPU + memory?

Process

- Abstraction of a running program
  - "a program in execution"
- Dynamic: has state, changes
  - Whereas a program is static



Basic Resources for Processes

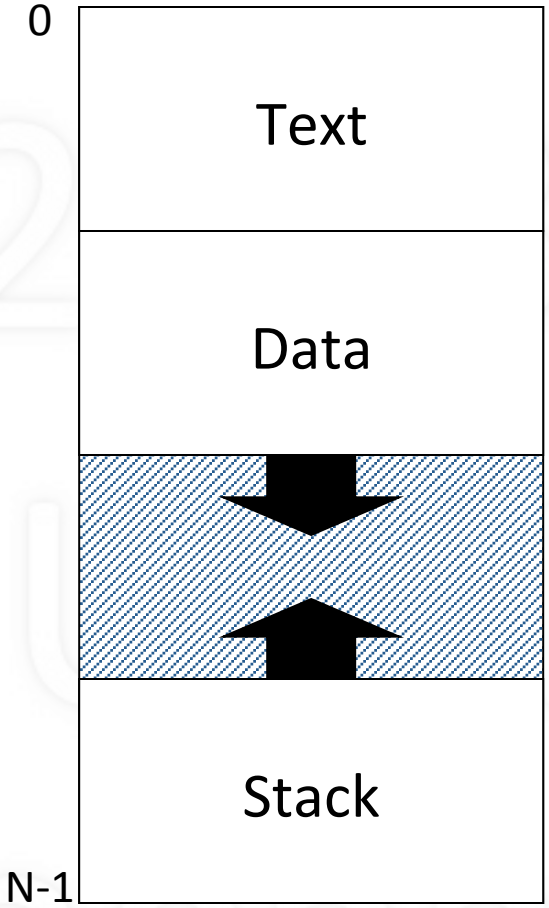
- CPU
  - Processing cycles (time)
  - To execute instructions
- Memory
  - Bytes or words (space)
  - To maintain state
- Other resources (e.g., I/O)

Context of a Process

- Context: machine and kernel-related state
- CPU context: values of registers
  - PC (program counter)
  - SP (stack pointer), FP (frame pointer), GP (general)
- Memory context: pointers to memory areas
  - Code, static variables (init, uninit), heap, shared, …
  - Stack of activation records
- Other (kernel-related state, …)

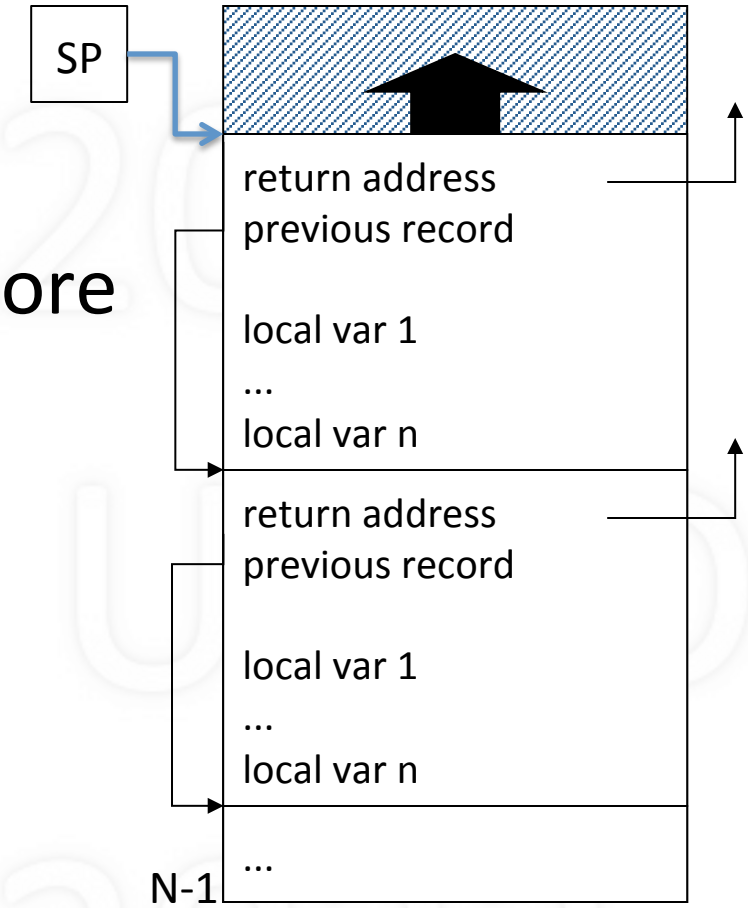
Process Memory Structure

- Text
  - Code: program instructions
- Data
  - Global variables
  - Heap (dynamic allocation)
- Stack
  - Activation records
  - Automatic growth/shrinkage

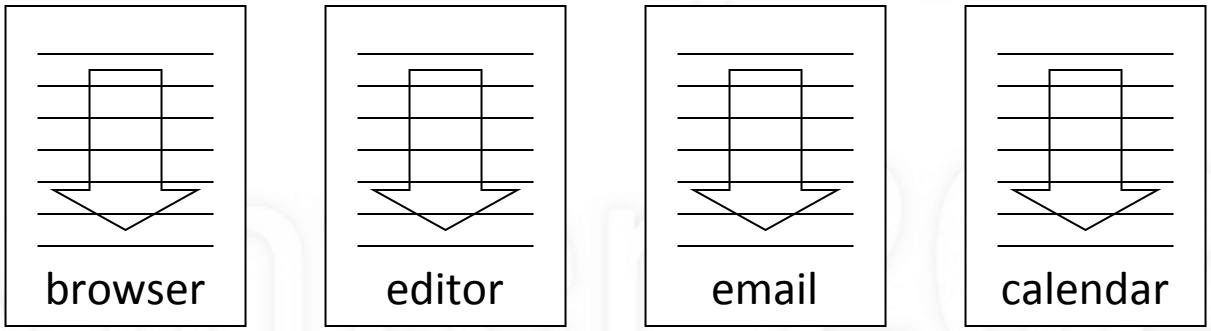


Process Stack

- Stack of activation records
  - One per pending procedure
- An activation record may store
  - where to return to
  - link to previous record
  - automatic (local) variables
  - other (e.g., register values)
- Stack pointer points to top



Goal: Support Multiple Processes



- Users would like to run multiple programs
  - "simultaneously"
  - Not all actively using the CPU
  - Some waiting for input, devices (e.g., disk), …
- How to do this given single CPU?

Multiprogramming

- Given a running process
  - At some point, it needs a resource, e.g., I/O device
  - Say resource is busy, process can't proceed
  - So, "voluntarily" gives up CPU to another process
- Yield (p)
  - Let process p run (voluntarily give up CPU to p)
  - Requires context switching

Context Switching

- Allocating CPU from one process to another
  - First, save context of currently running process
  - Next, restore (load) context of next process to run
- Loading the context
  - Load general registers, stack pointer, etc.
  - Load program counter (must be last instruction!)

Simple Context Switching

- Two processes: A and B
- A calls Yield(B) to voluntarily give up CPU to B
- Save and restore registers
  - General-purpose, stack pointer, program counter
- Switch text and data
- Switch stacks
  - Note that PC is in the middle of Yield!

The “magic” of Yield

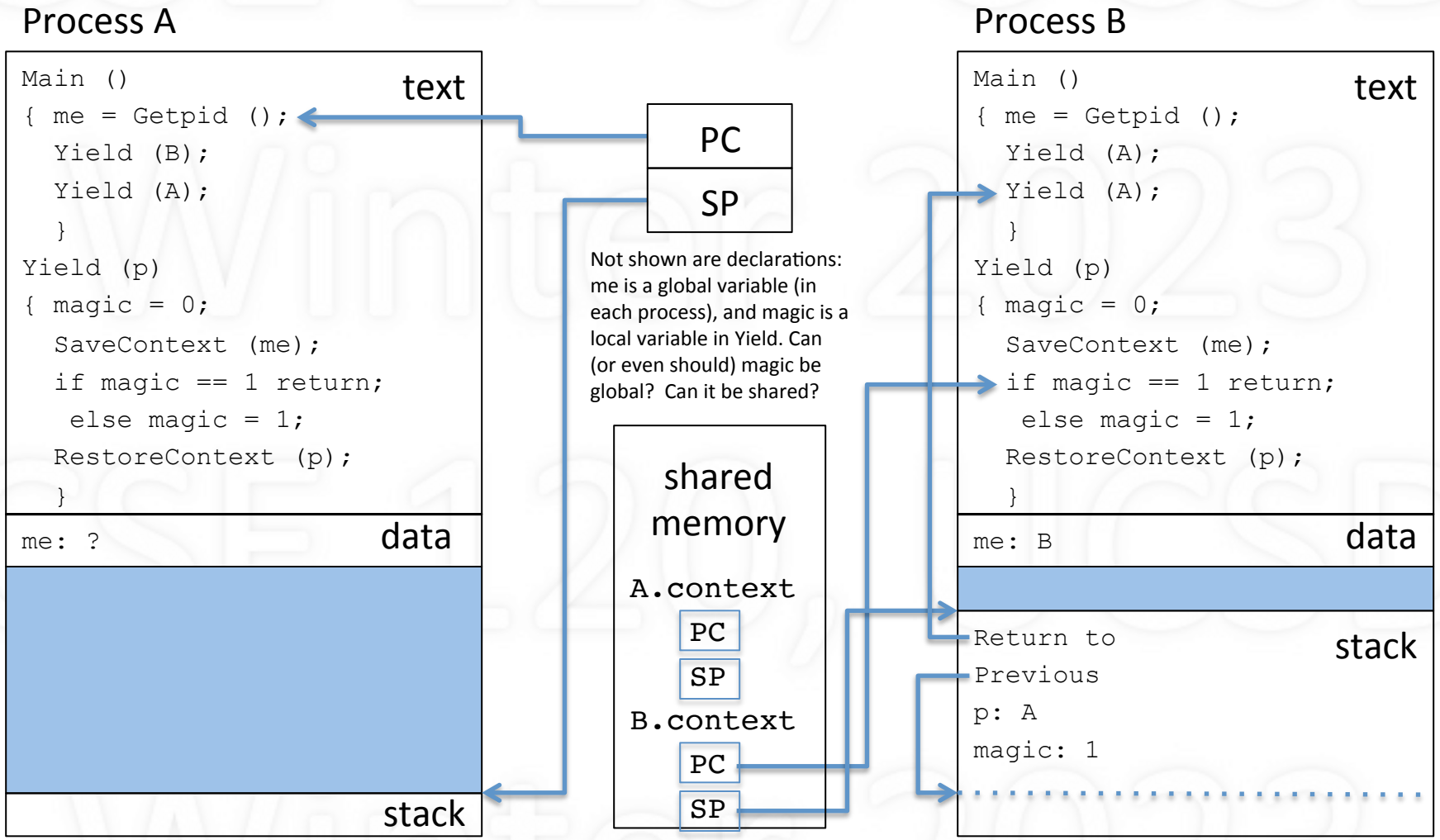
```
magic = 0;              // local variable
save A’s context:       // current process
   asm save GP;         // general purpose registers;
   asm save SP;         // stack pointer
   asm save PC;         // program counter, note value!
if (magic == 1) return;
else magic = 1;
restore B’s context: // process being yielded to
   asm restore GP;
   asm restore SP;
   asm restore PC;      // must be last!
```

Example

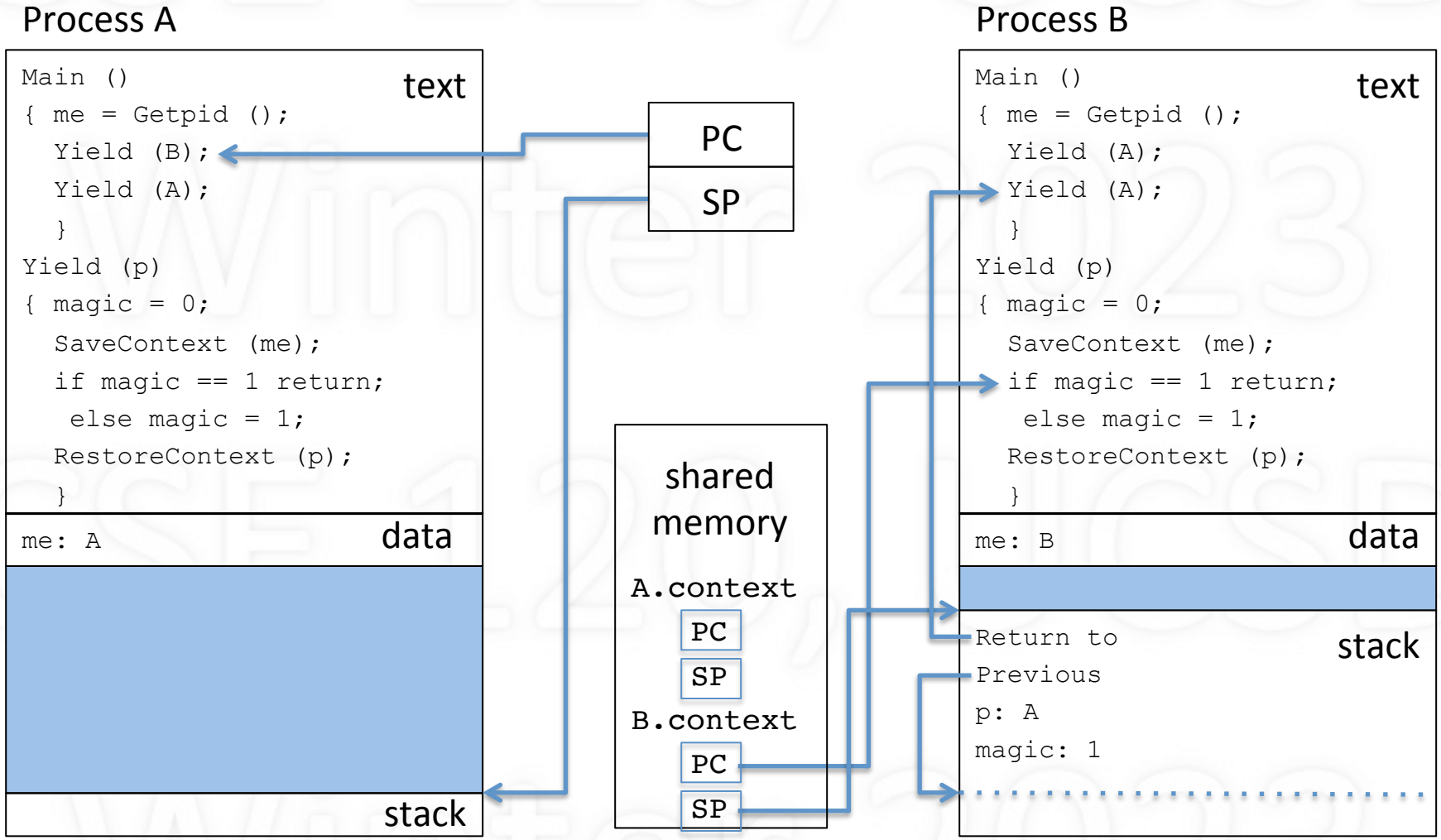
```
int me;

Main ()  /* process A */
{
   me = Getpid ();
   Yield (B);
   Yield (A);
}

Yield (p)
{   int magic;

   magic = 0;
   SaveContext (me);
   if magic == 1 return;
    else magic = 1;
   RestoreContext (p);
}
```

```
int me;

Main ()  /* process B */
{
    me = Getpid ();
    Yield (A);
    Yield (A);
}

Yield (p)
{   int magic;

    magic = 0;
    SaveContext (me);
    if magic == 1 return;
     else magic = 1;
    RestoreContext (p);
}
```

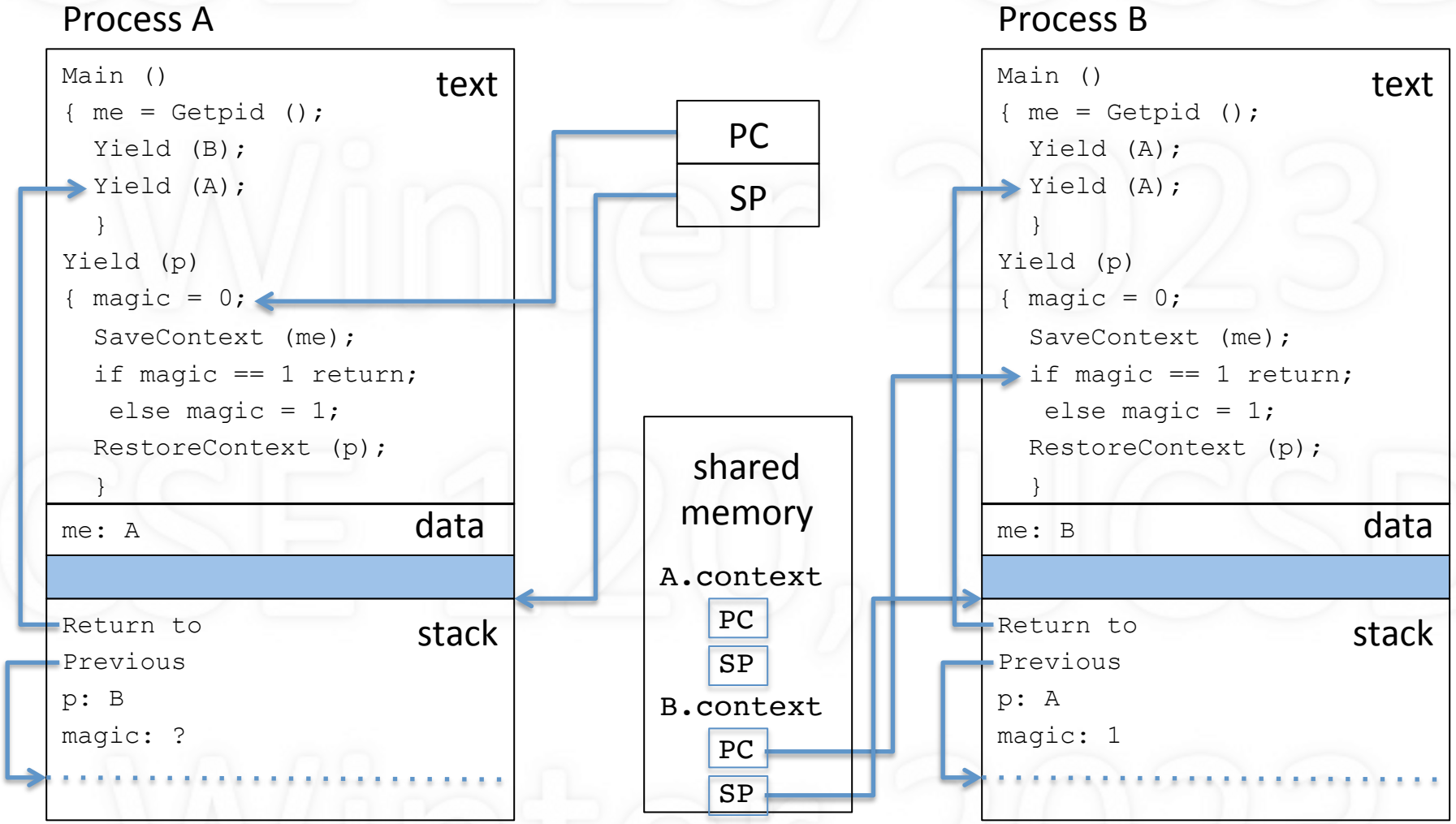
In this example, A is about to set me to its process ID, and yield to B. B had already yielded to A: note B's saved PC and SP.



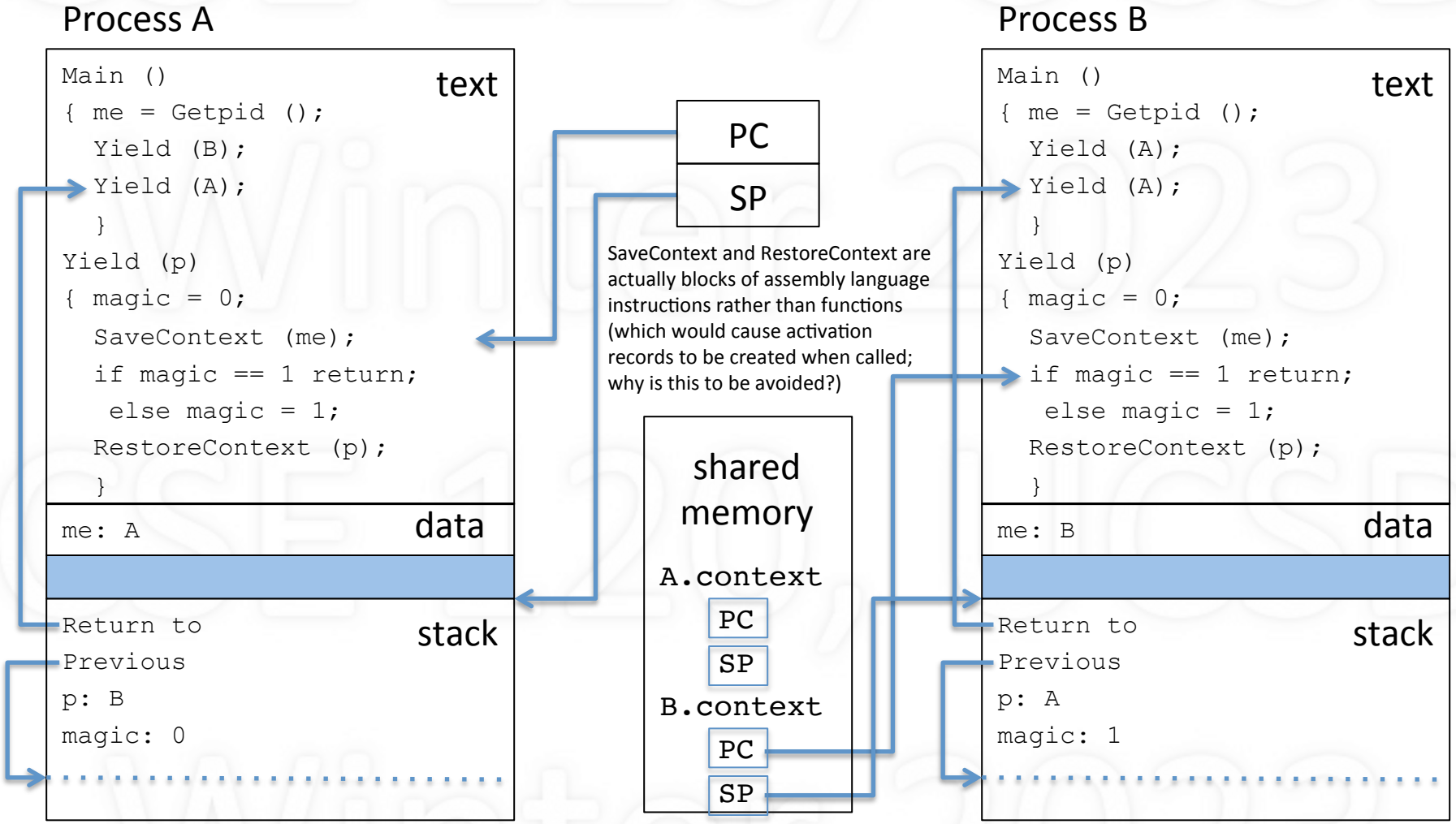
Process A has just set me to ‘A’ and is about to call Yield. The PC always points to the next instruction to be executed.



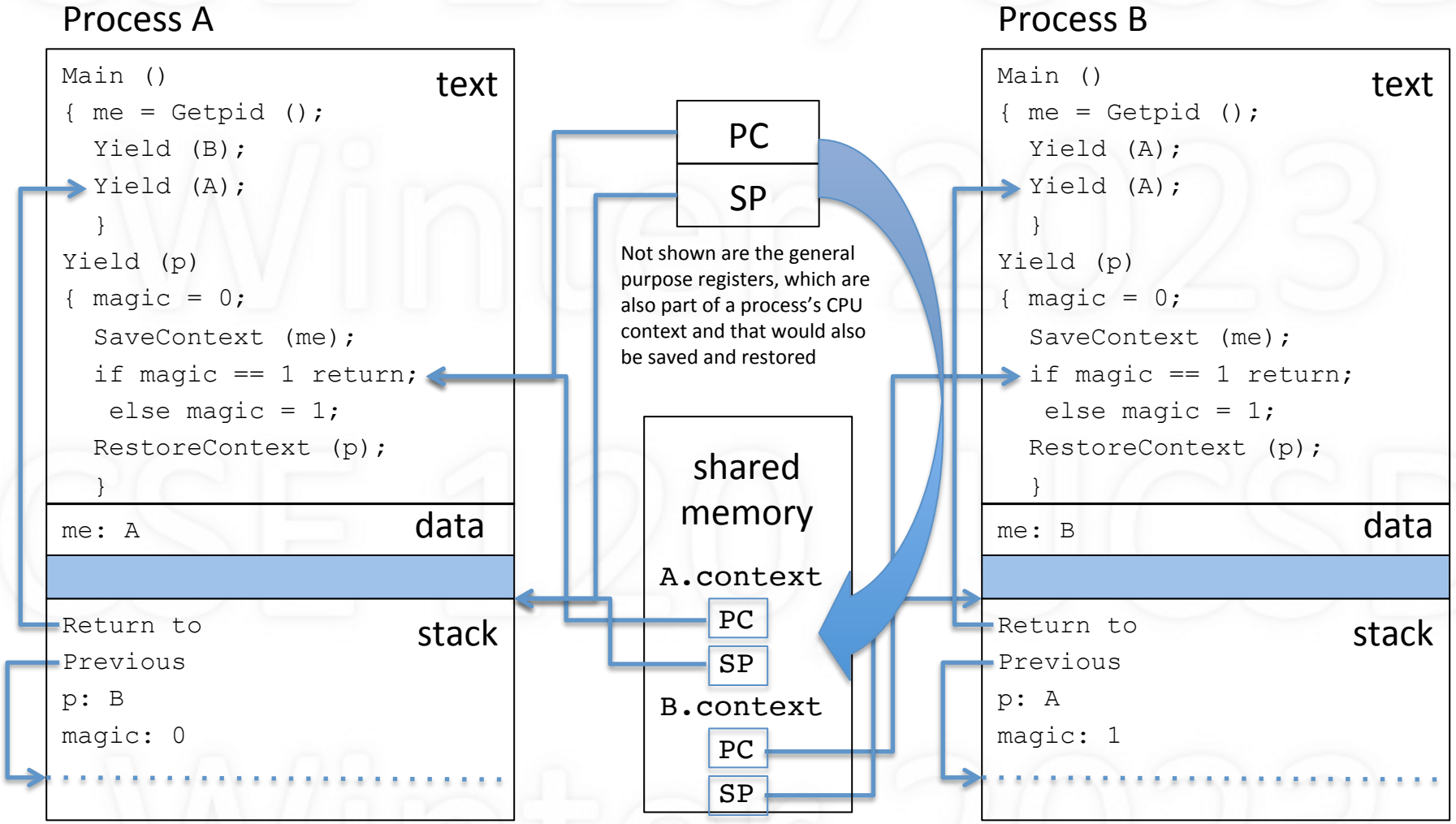
Upon entering Yield, an activation record is pushed on the stack. It contains links, and local variables p and magic.



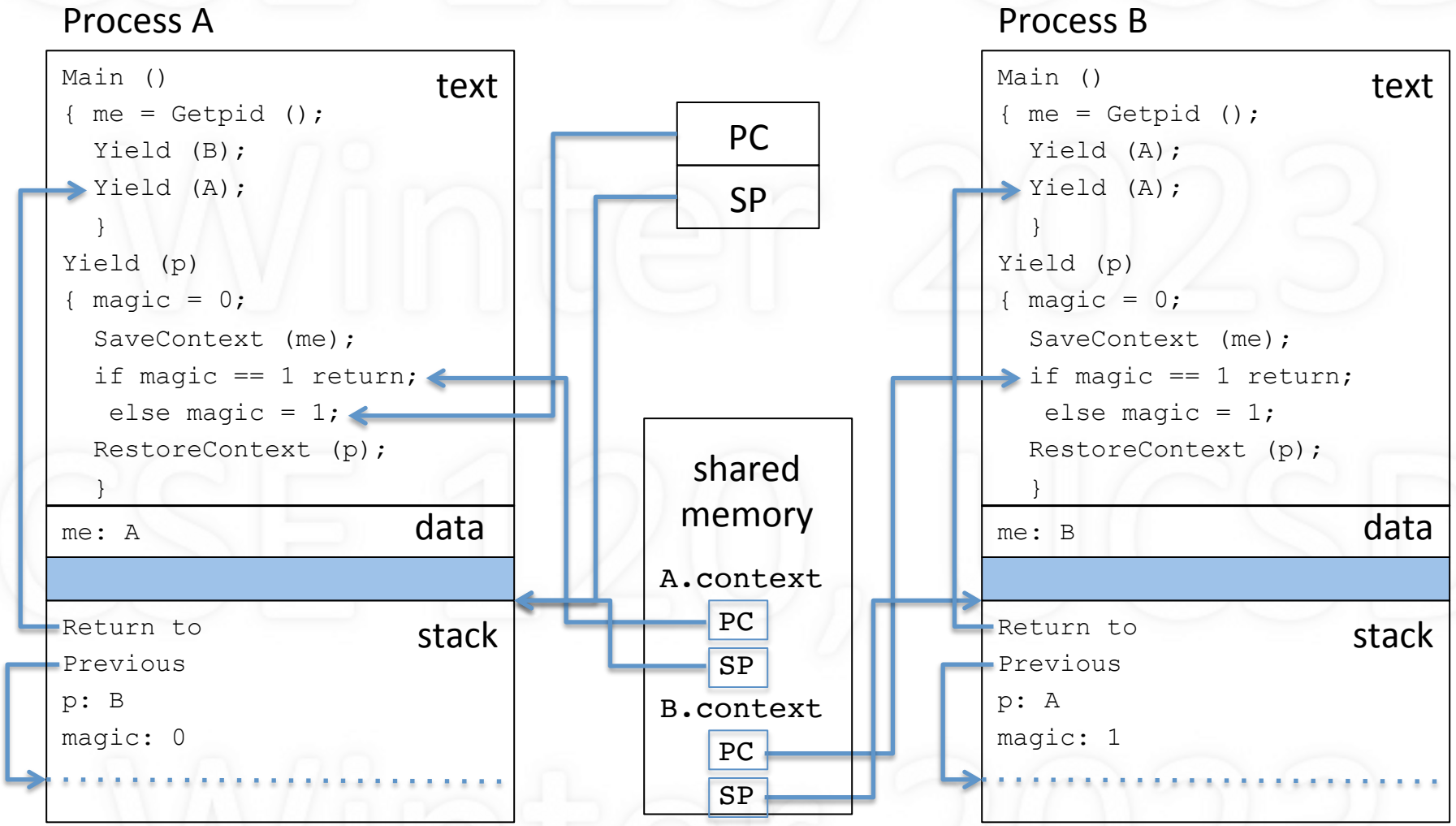
Variable magic is set to 0. It is an automatic variable, dynamically allocated on the stack. Next: SaveContext.



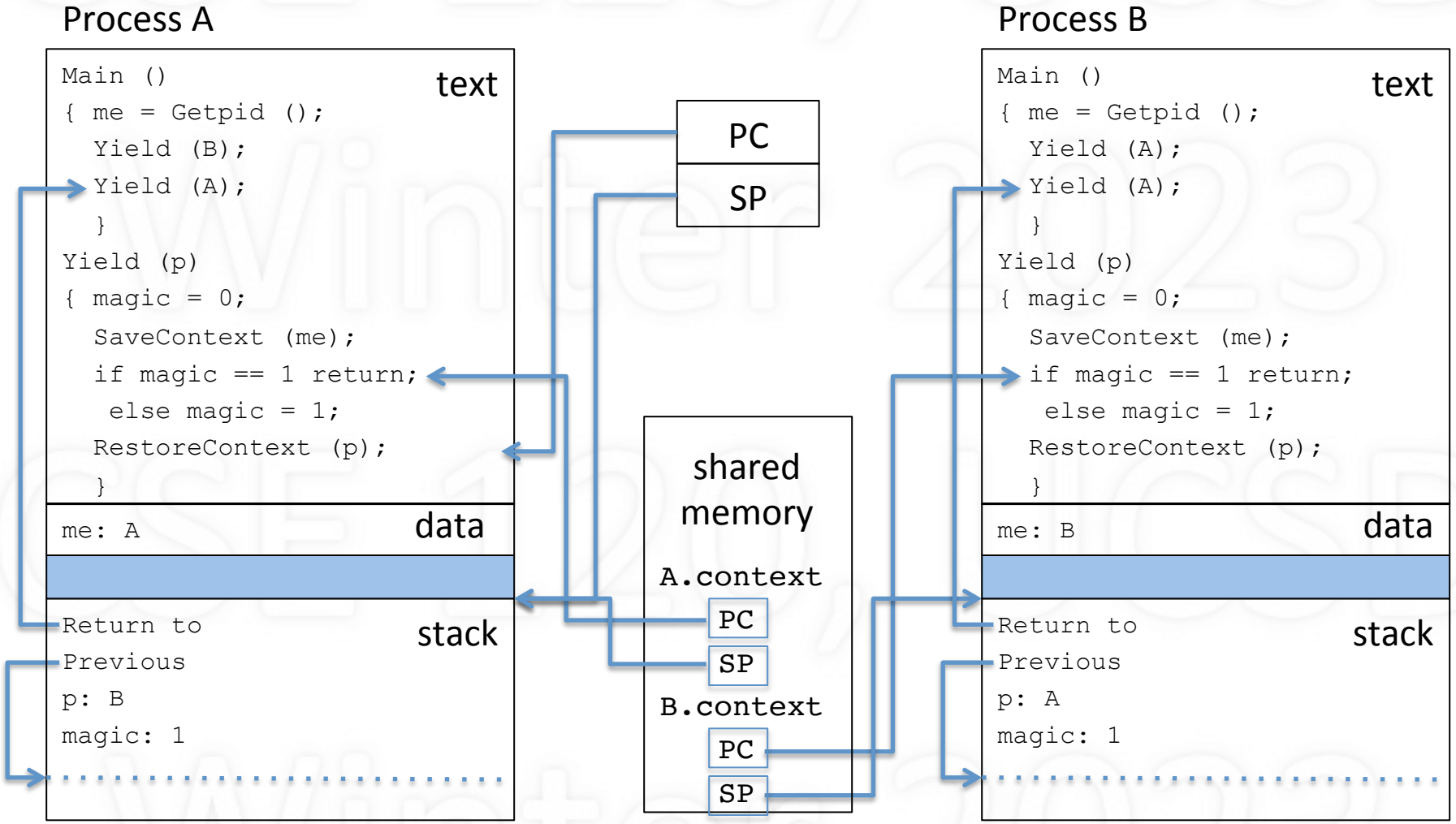
A's context is now saved. The saved PC points to just after the call to SaveContext. Compare this to B's saved context.



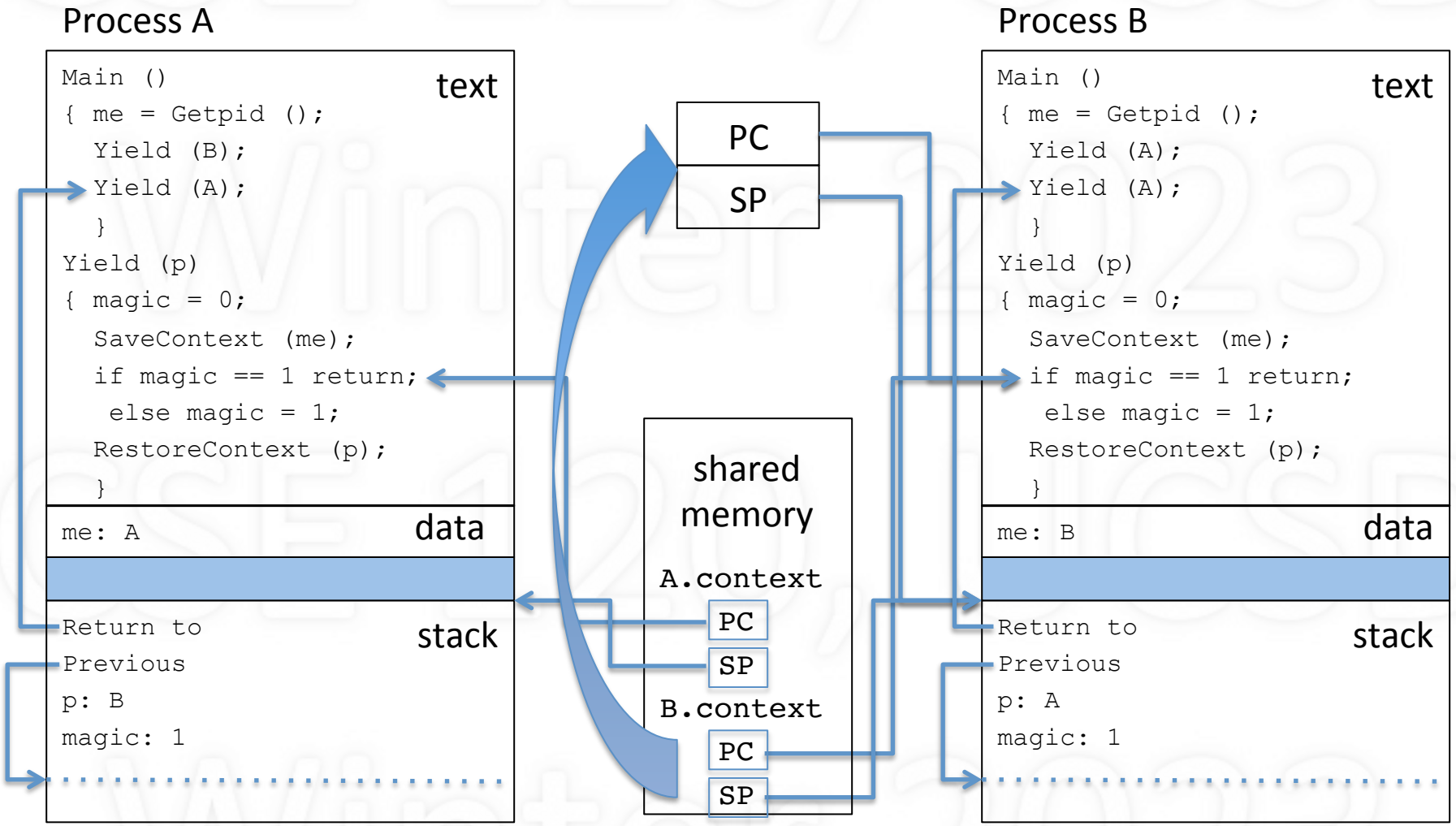
A just checked whether magic equals 1, which was false, and so, on to the else clause to set magic to 1.



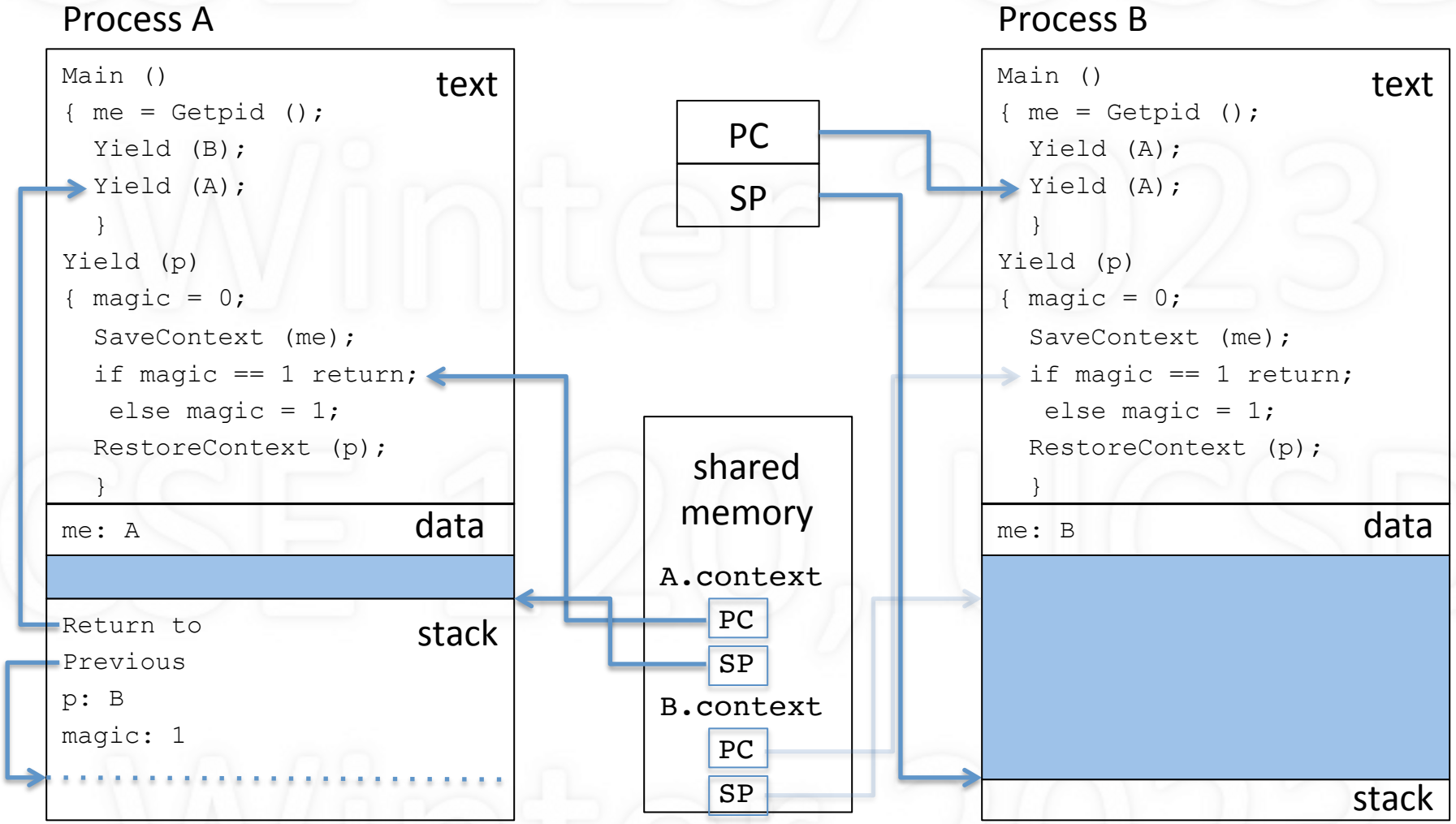
A sets magic to 1, and is about to restore B's context (just like B's situation in the past when it restored A's context).



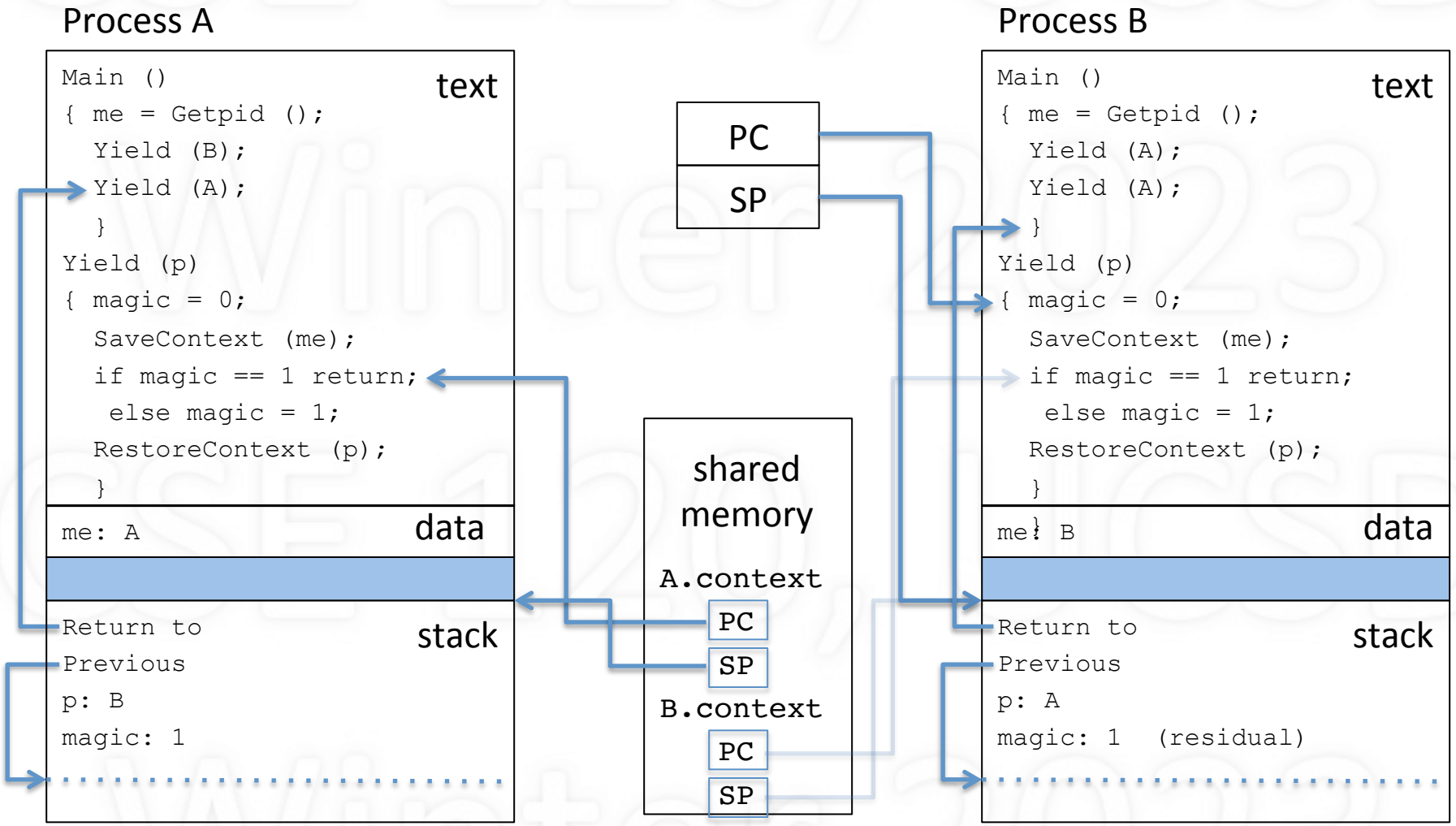
B's context is restored (i.e., the machine state is now that of B). The PC points to the if statement.



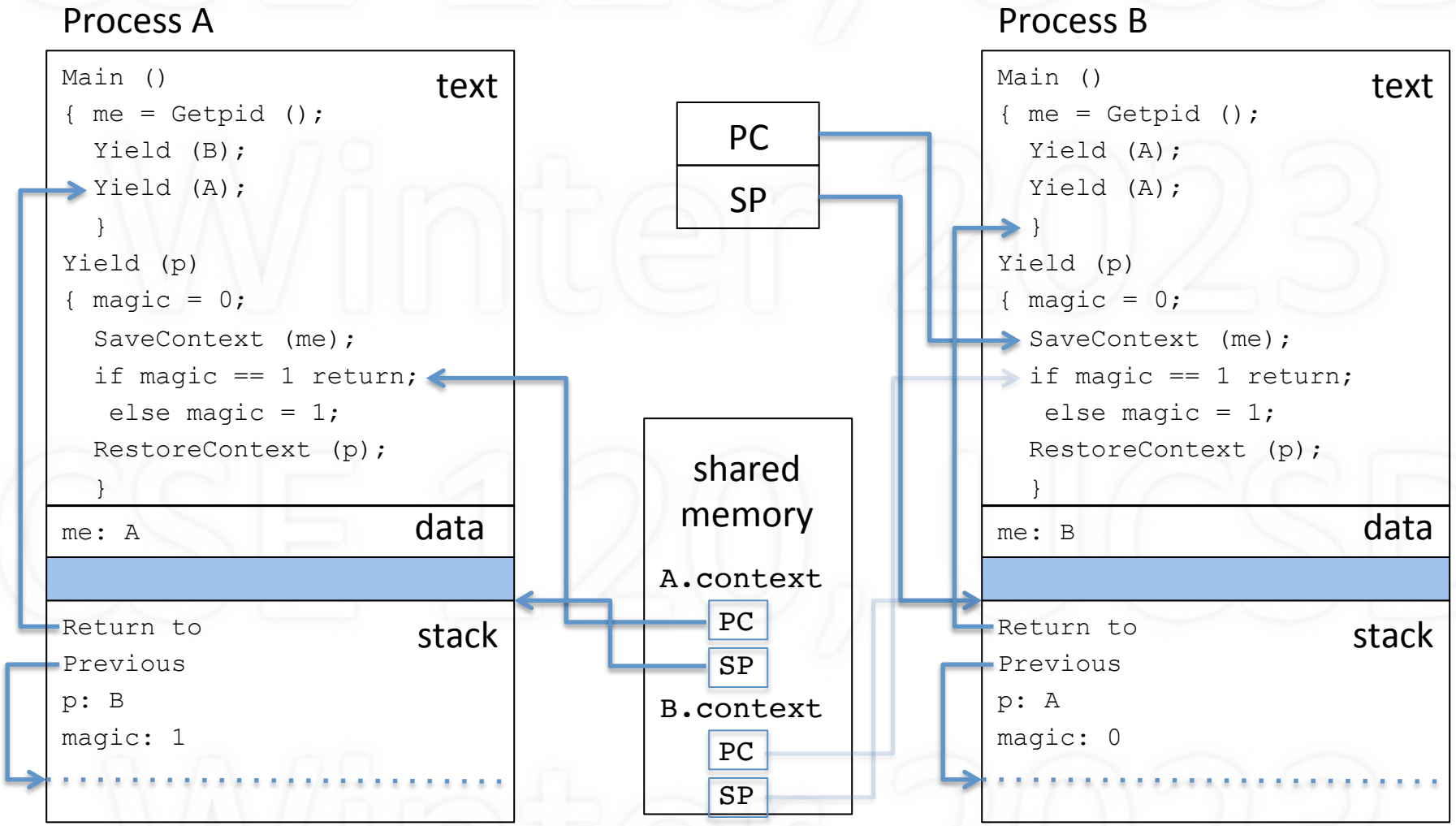
Since magic equals 1, B returns from Yield (unlike last time when magic equaled 0). No wonder it's called magic!



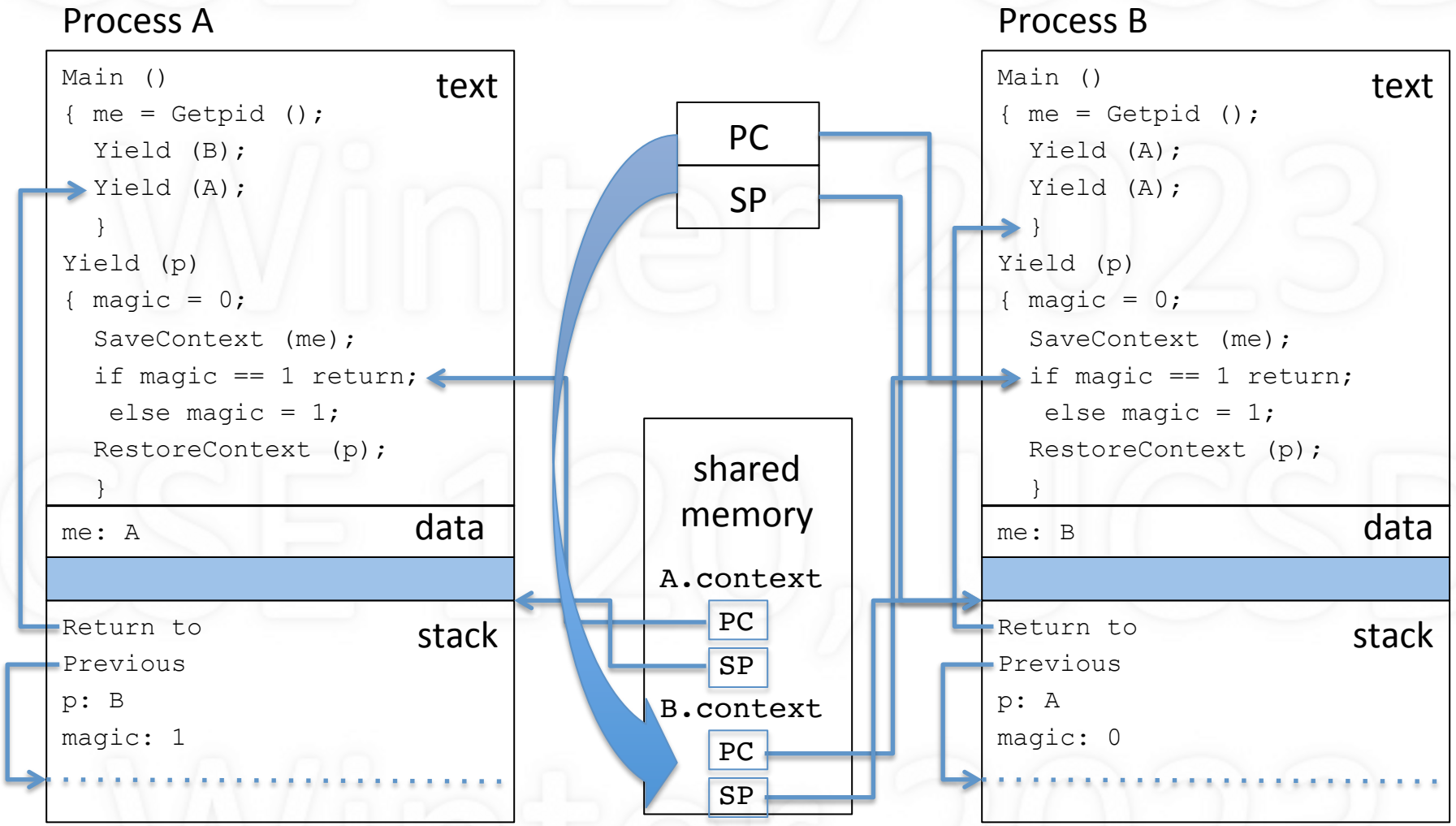
See if you can figure out how the rest of this works.



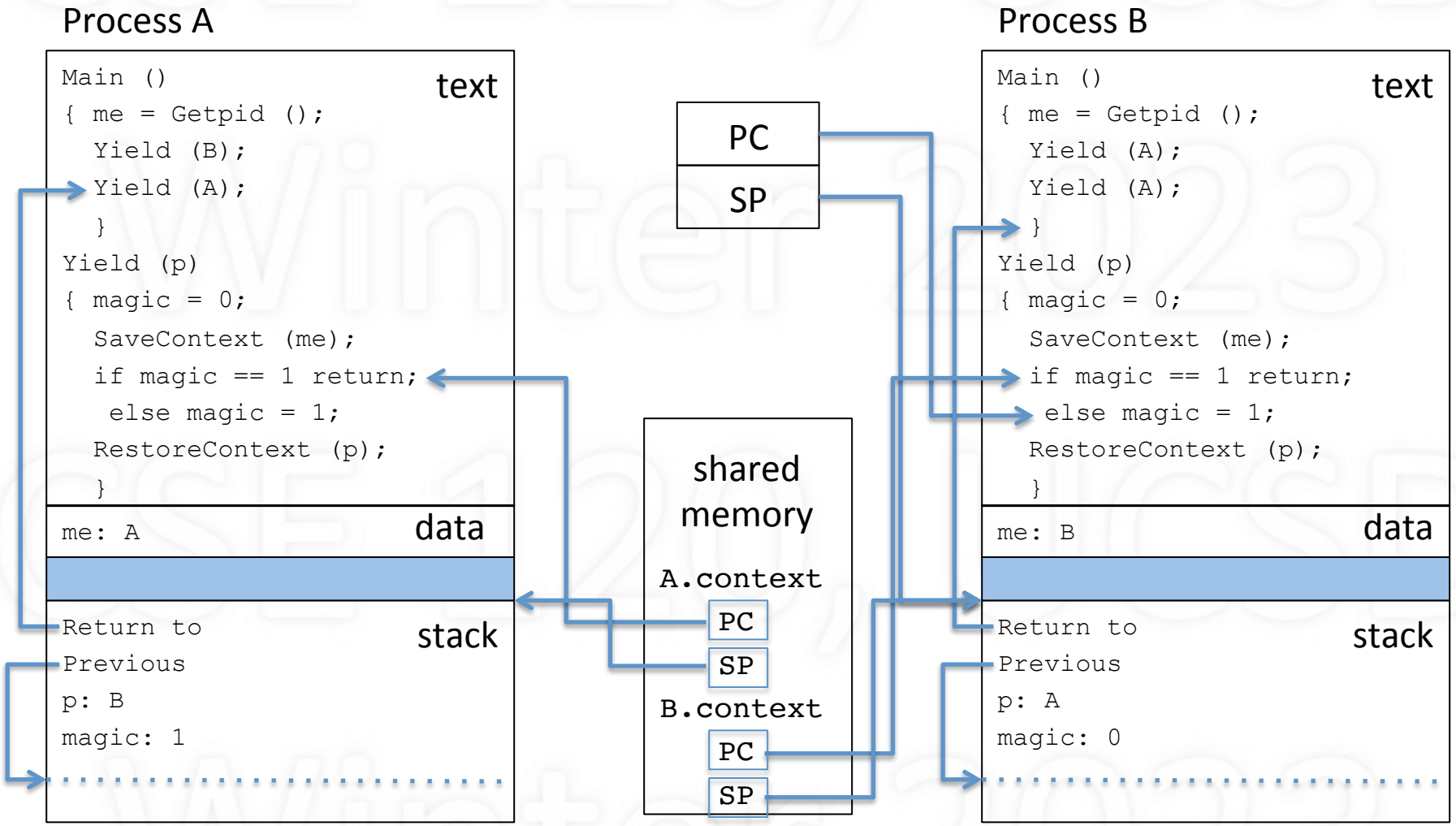
B is yielding to A.



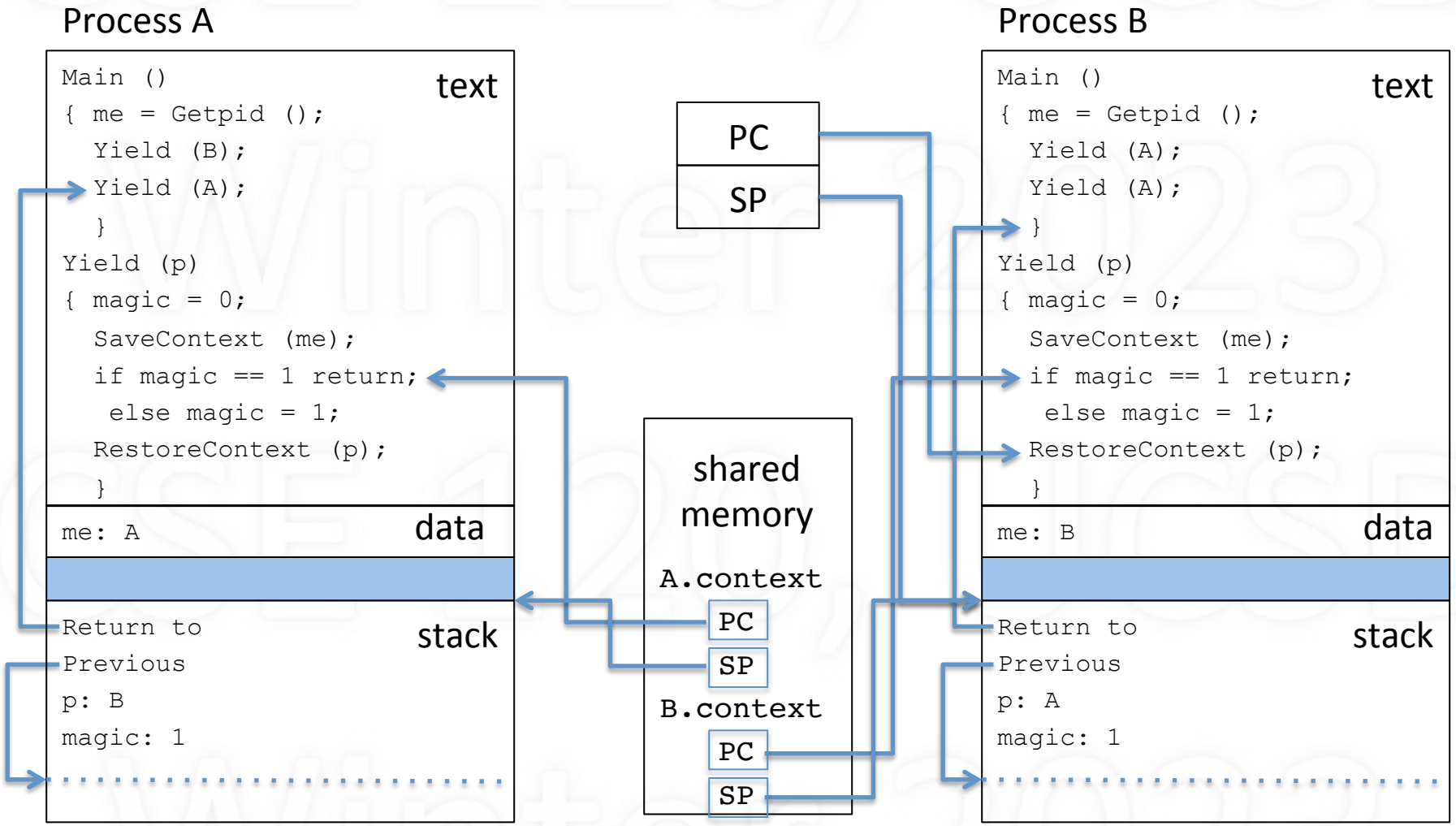
Goes through the same steps as we saw for A.



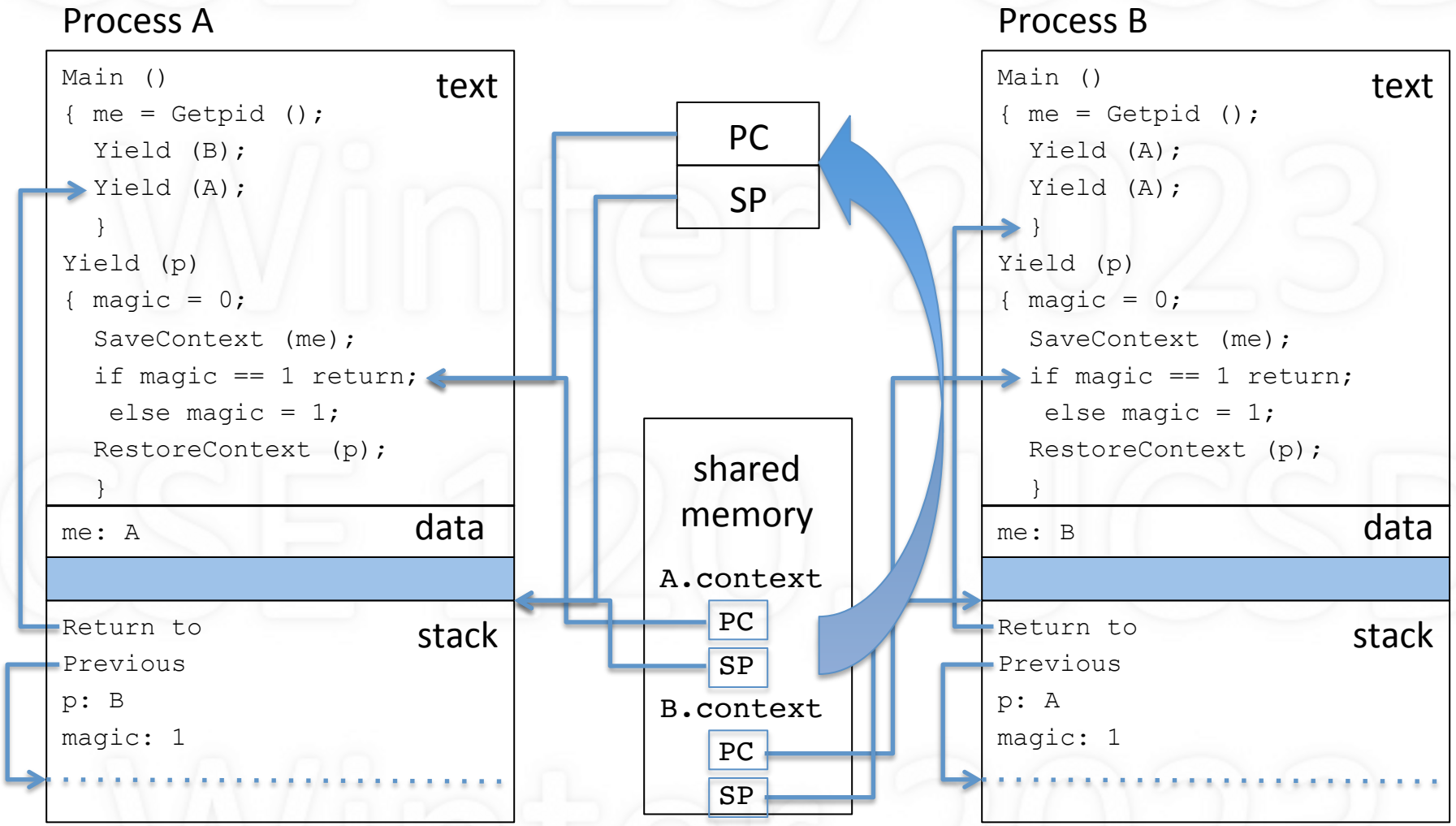
Note value of magic.



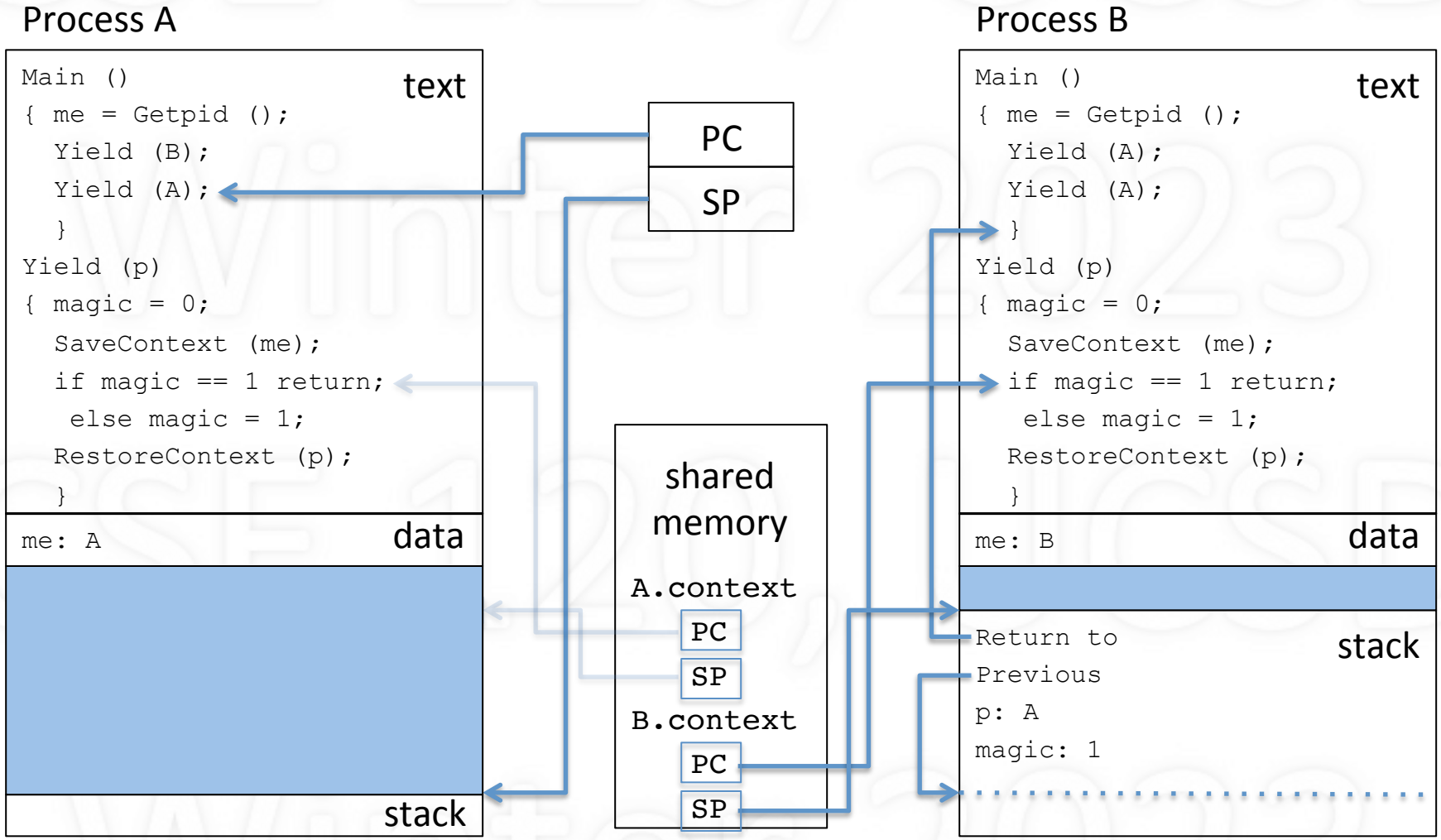
About to restore context of A.



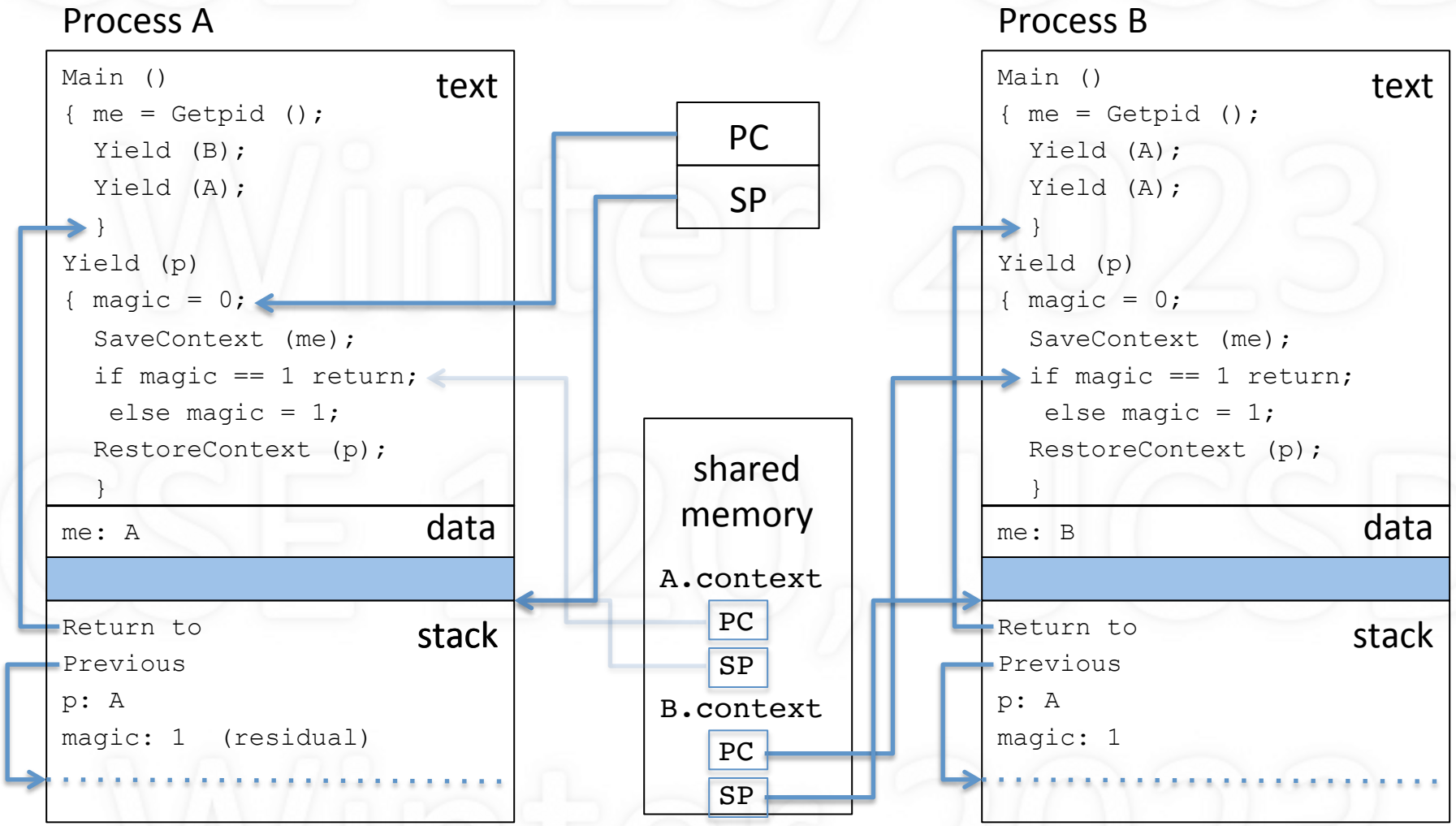
A resumes. Note value of magic.



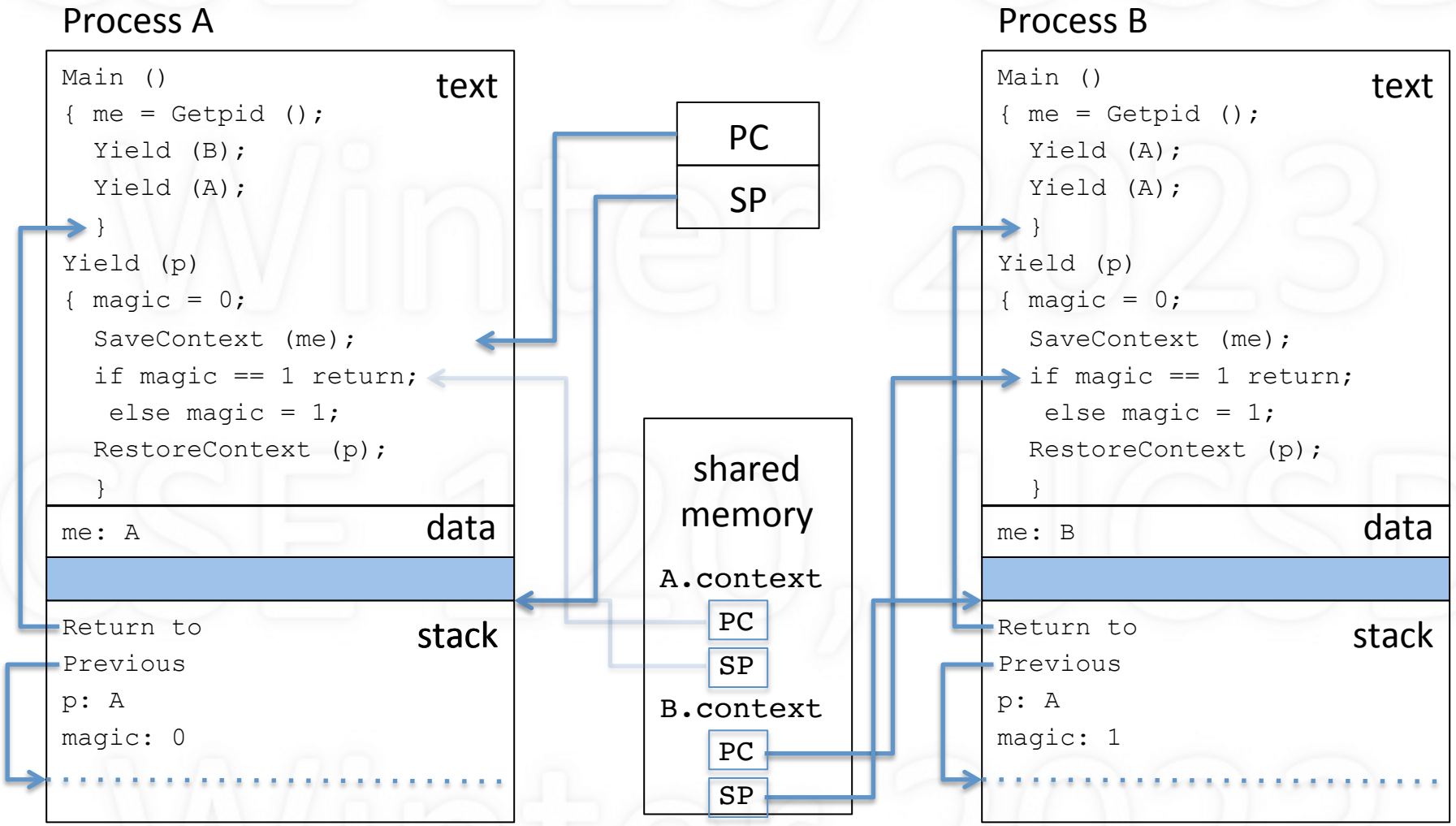
Returns from Yield, and now about to call Yield again, but to itself!



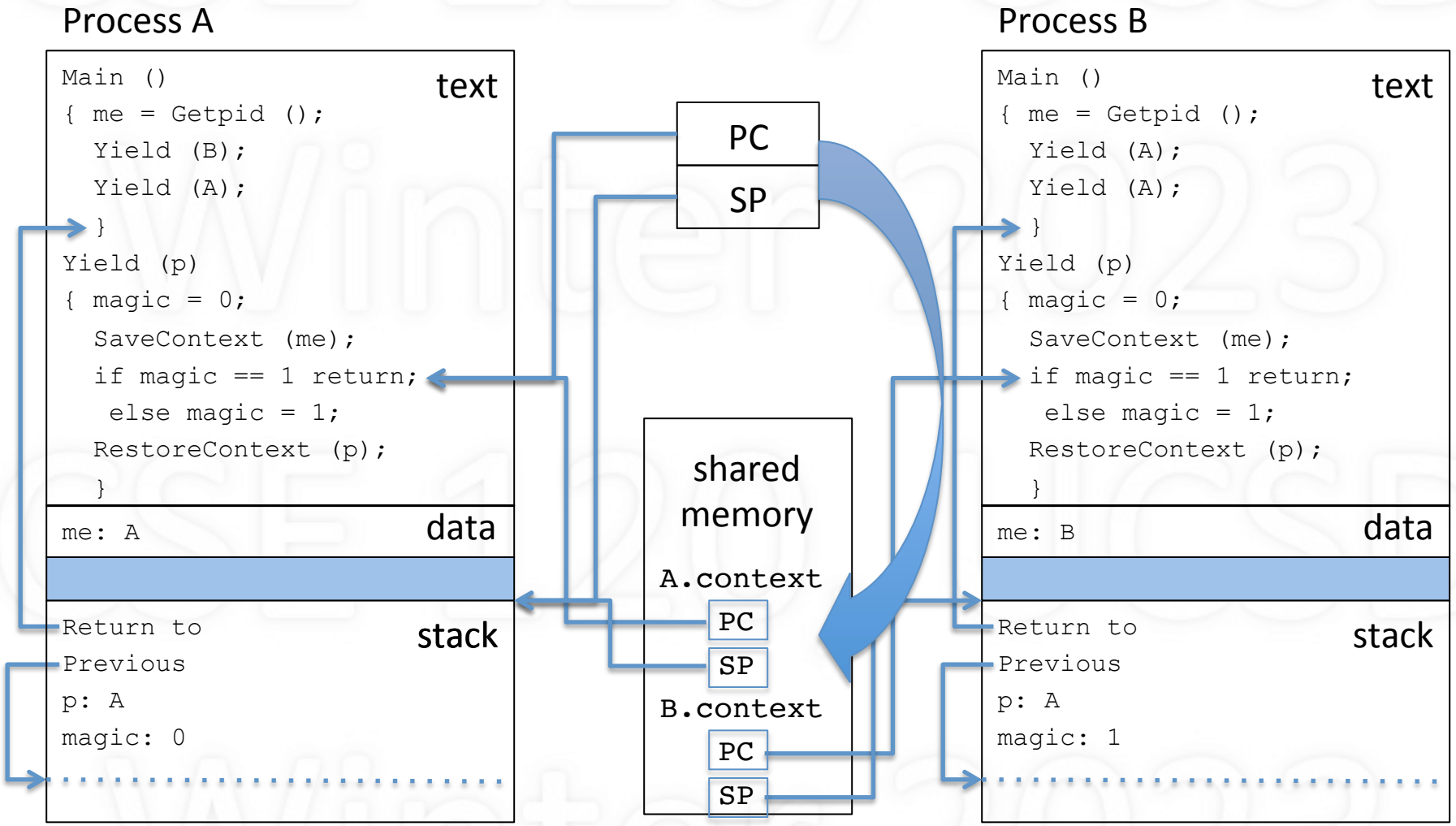
Inside Yield. Note new activation record on stack.



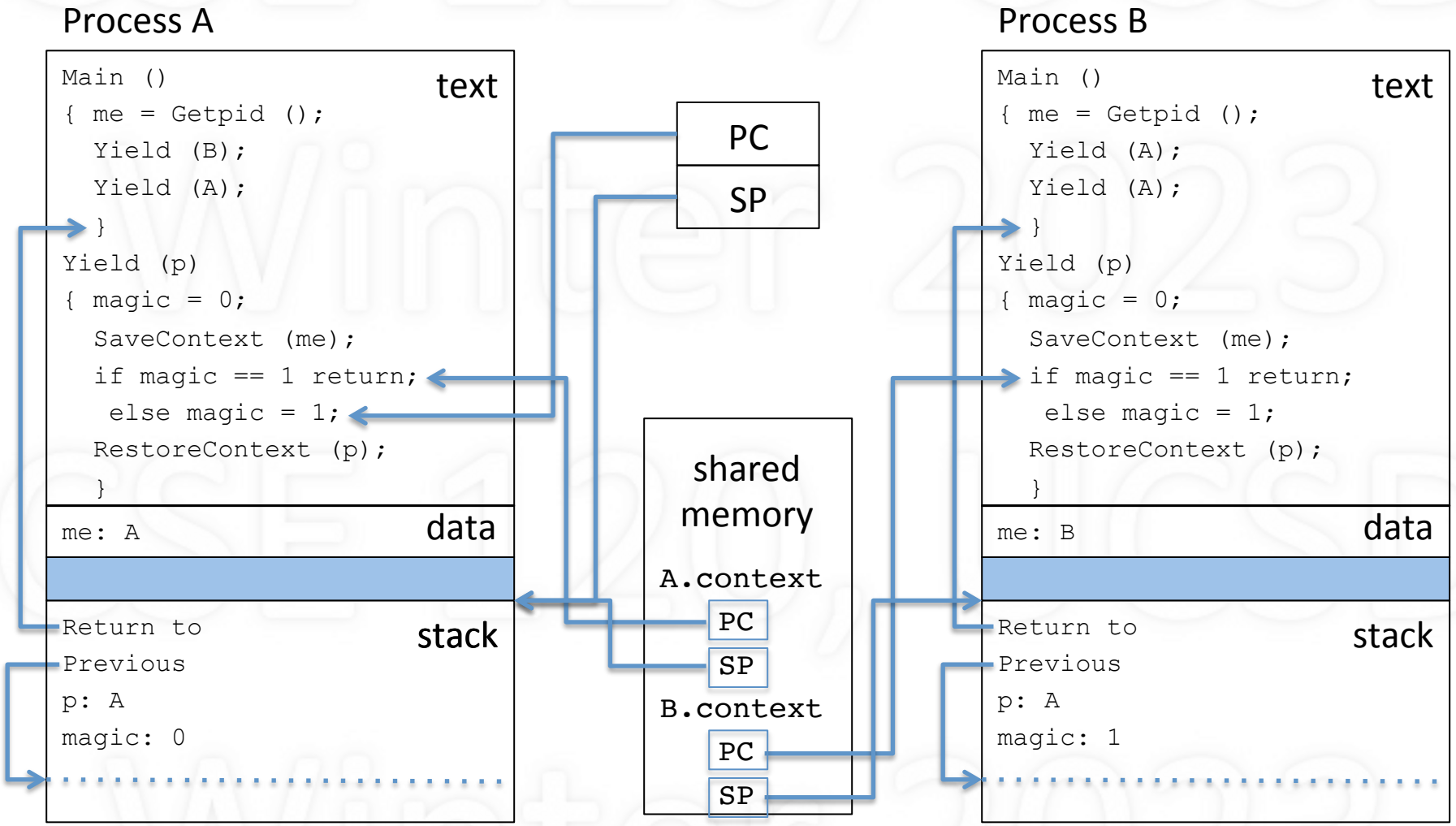
About to save context of A.



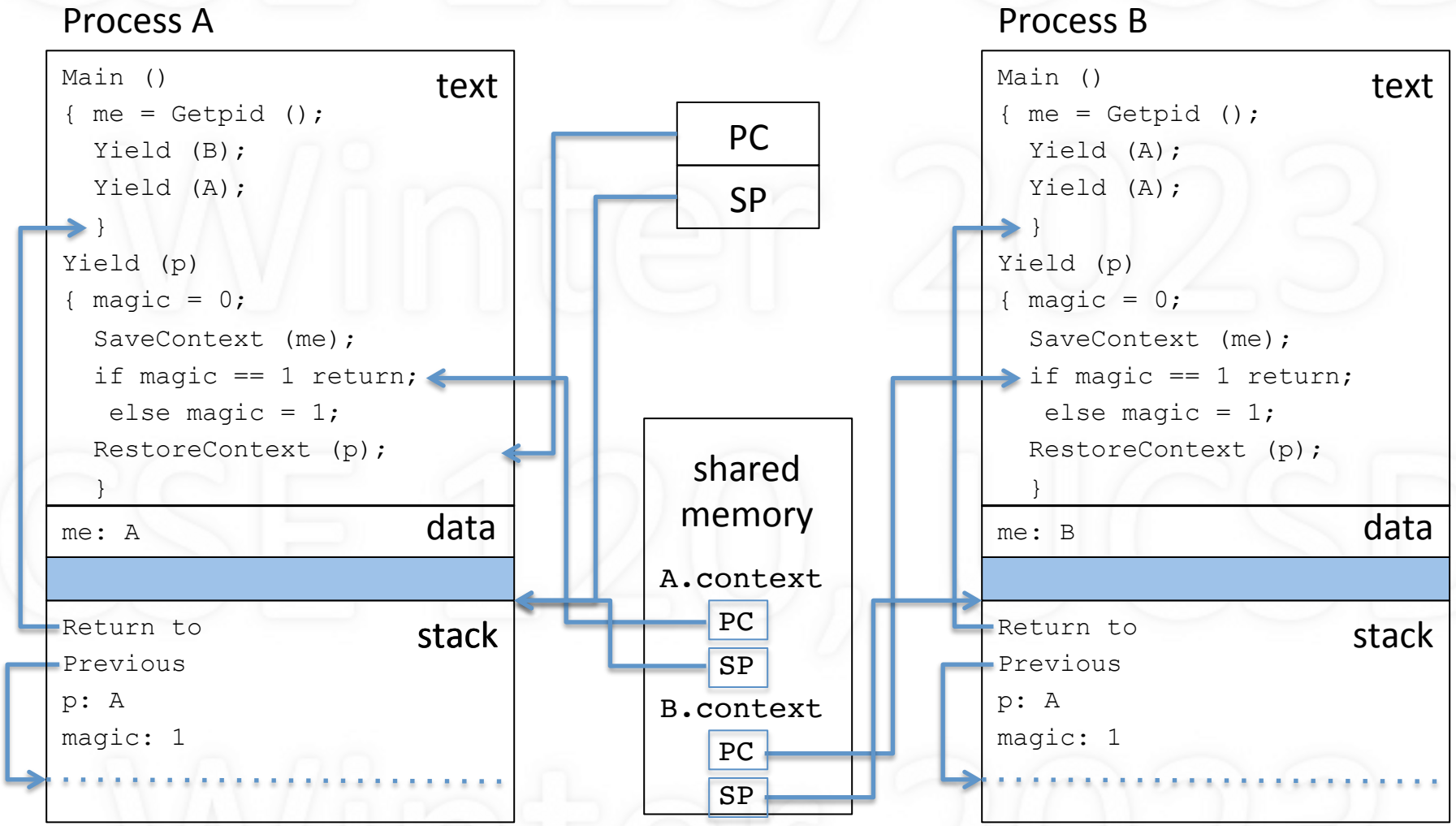
Note value of magic.



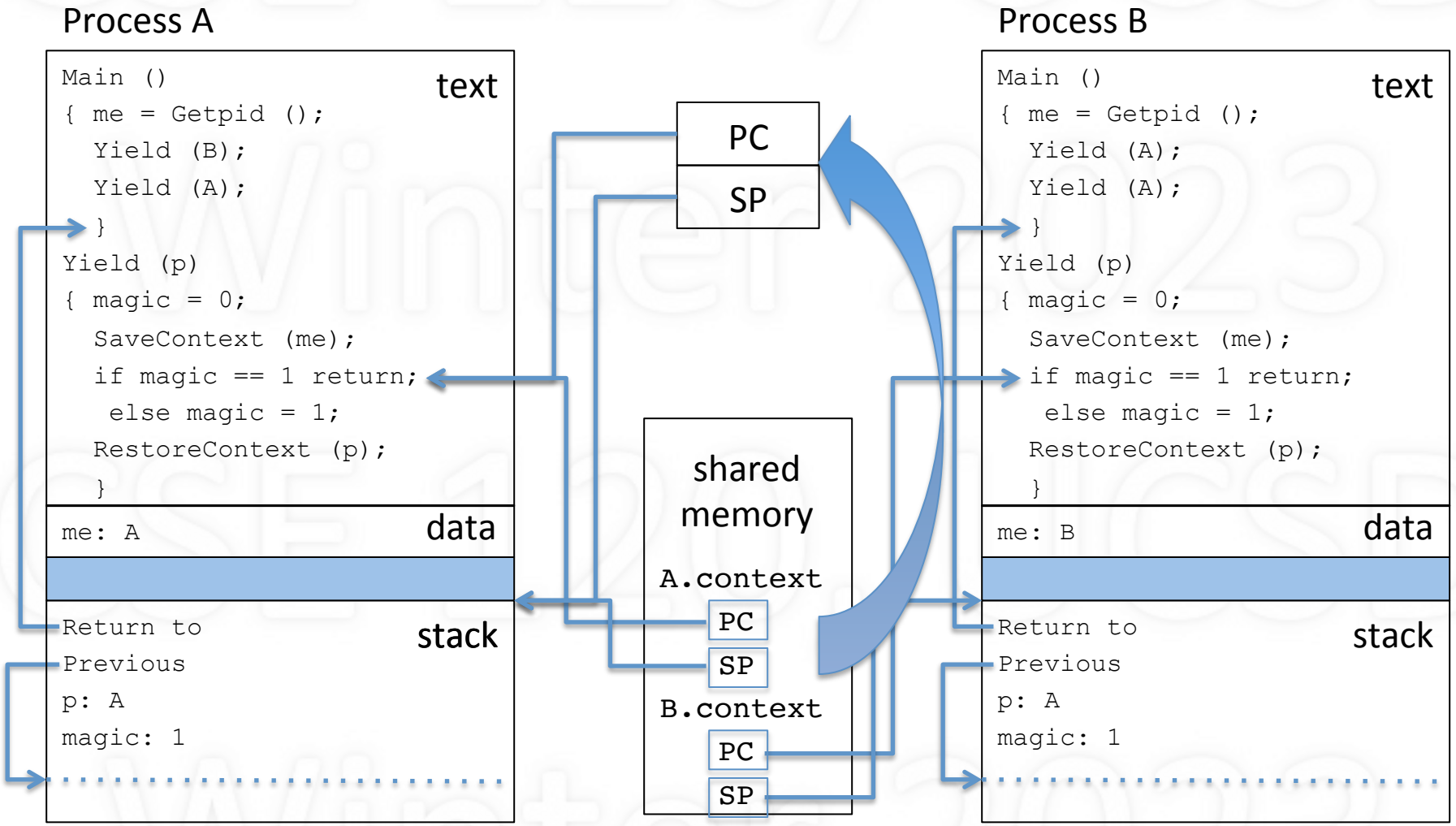
Set magic.



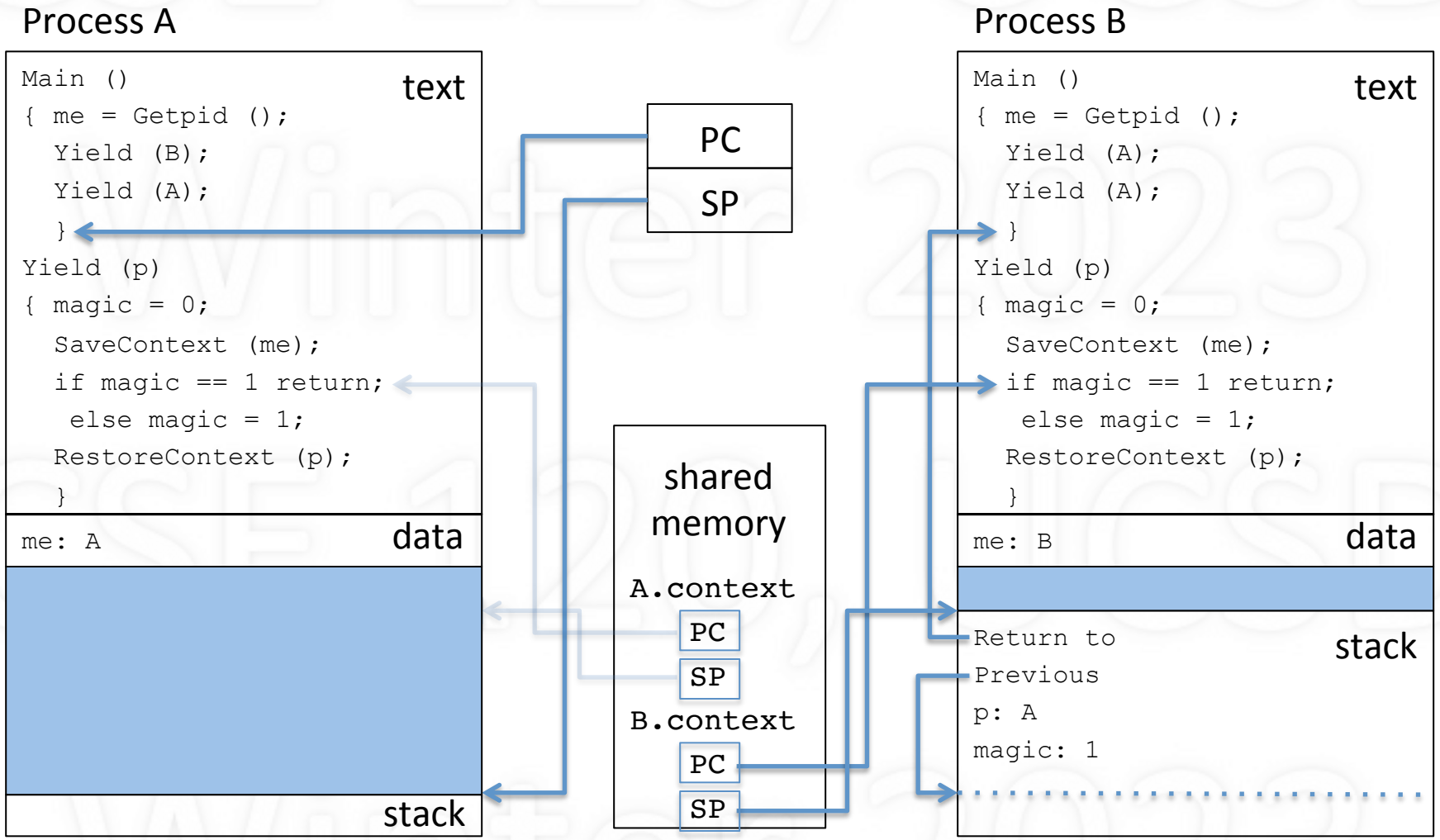
Restore context of A!



Note where A executes.



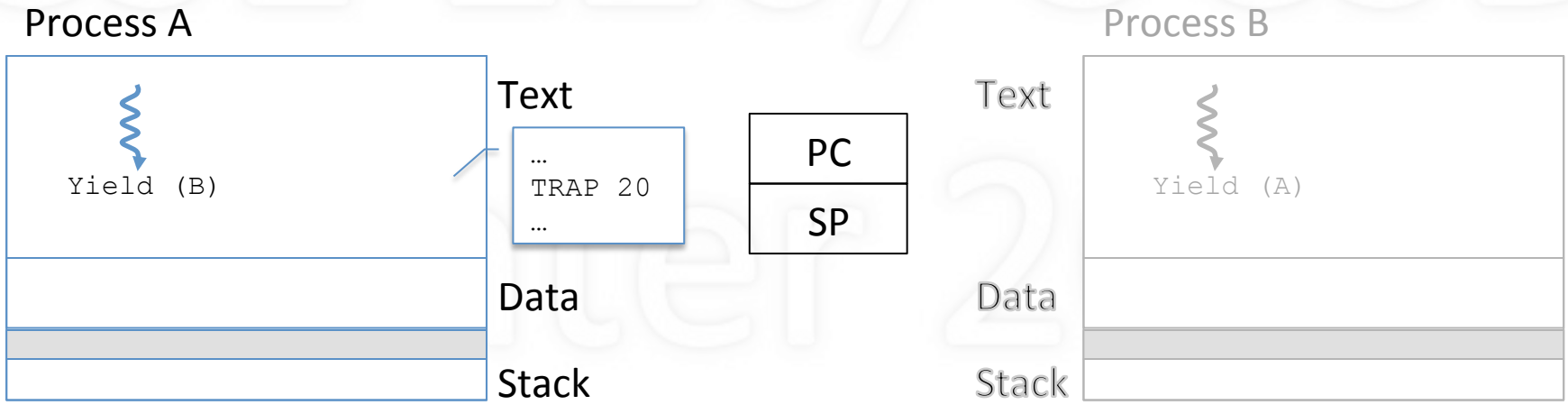
Return from Yield.



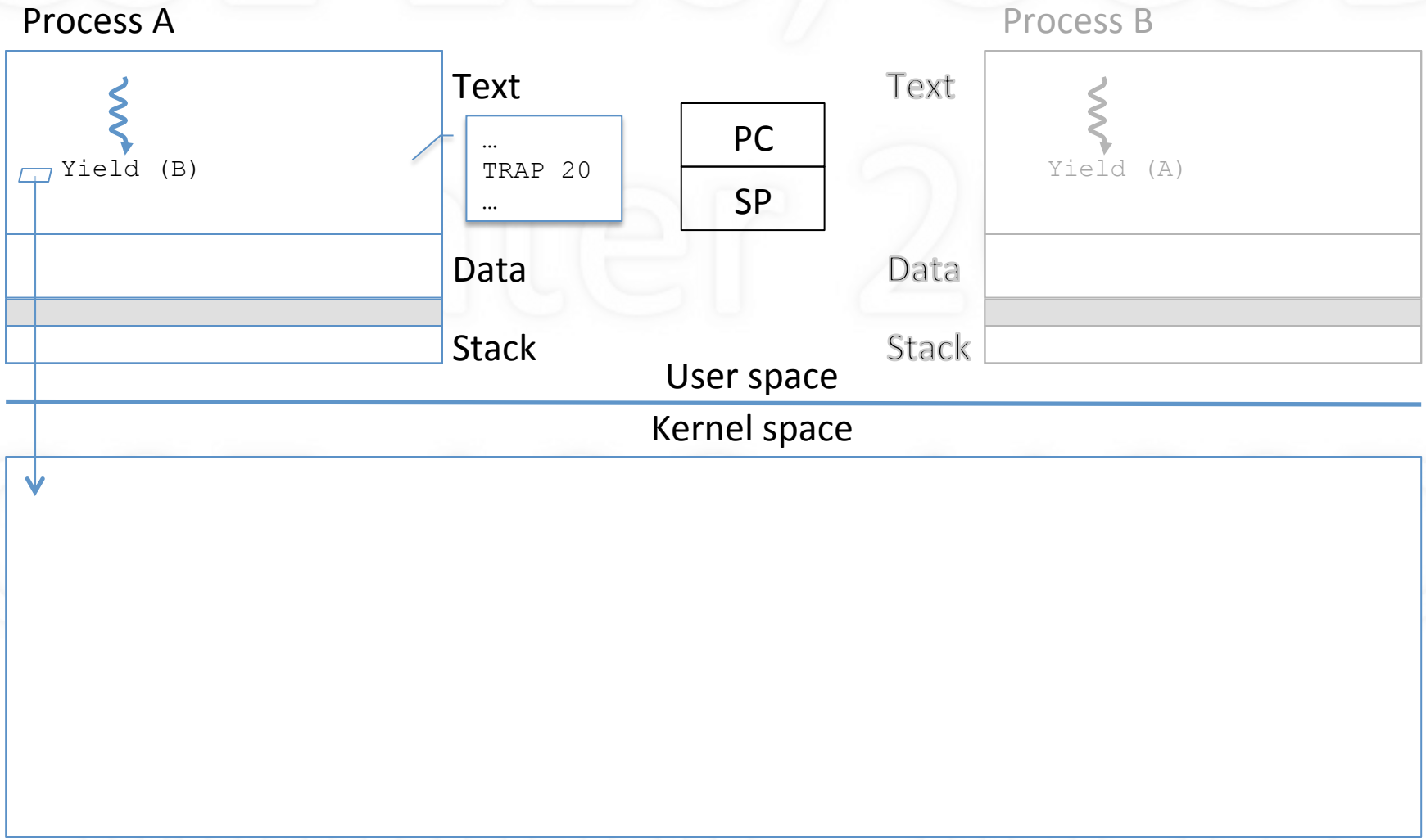
Yielding via the Kernel

- Yield routine is common code: put in kernel
- Process contexts are also in the kernel
  - This way they are protected
  - Only needed by Yield routine anyway
- But what is the kernel?
  - code that supports processes
  - runs as an extension of current process
- Has text, data, and multiple stacks

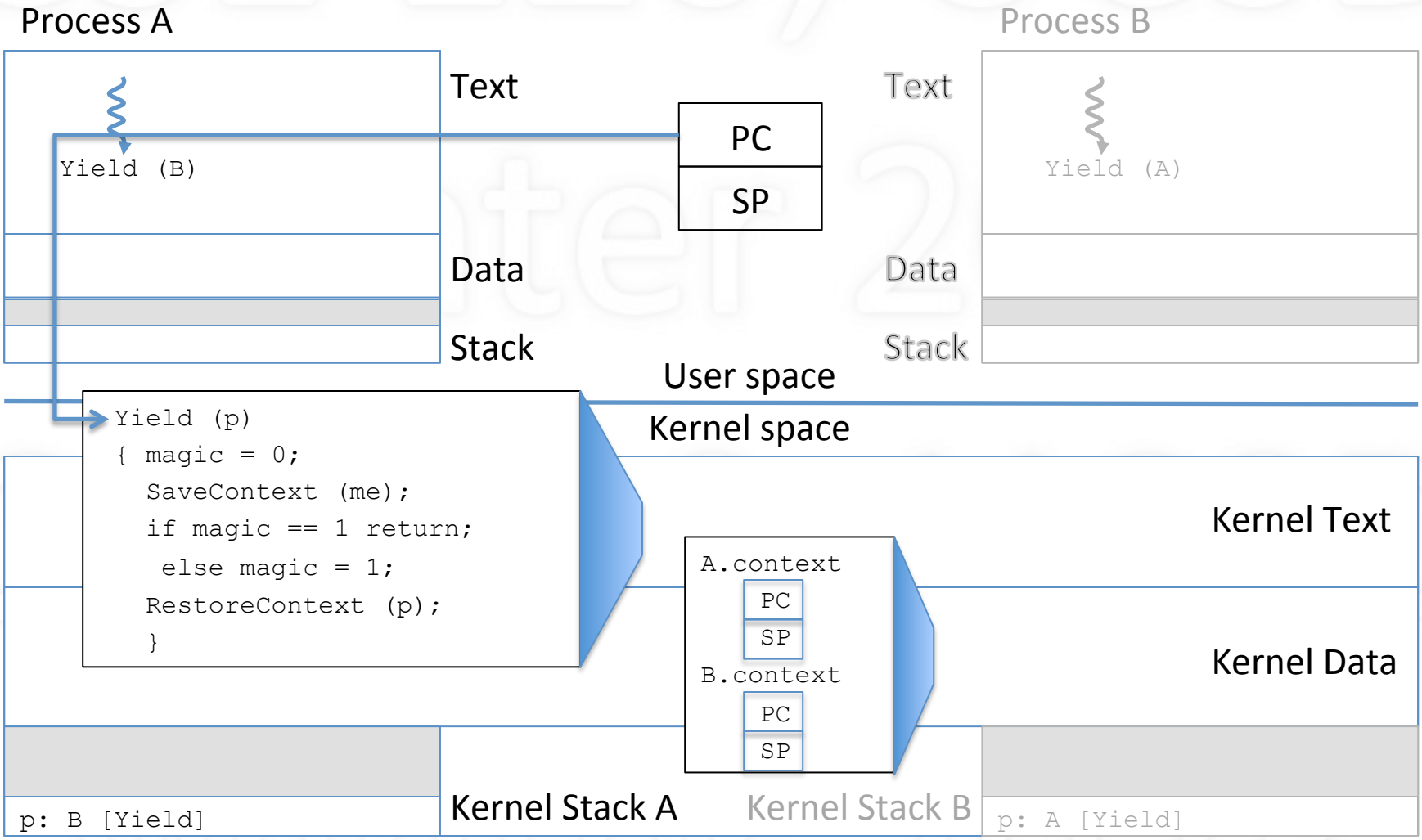
Yielding via the Kernel



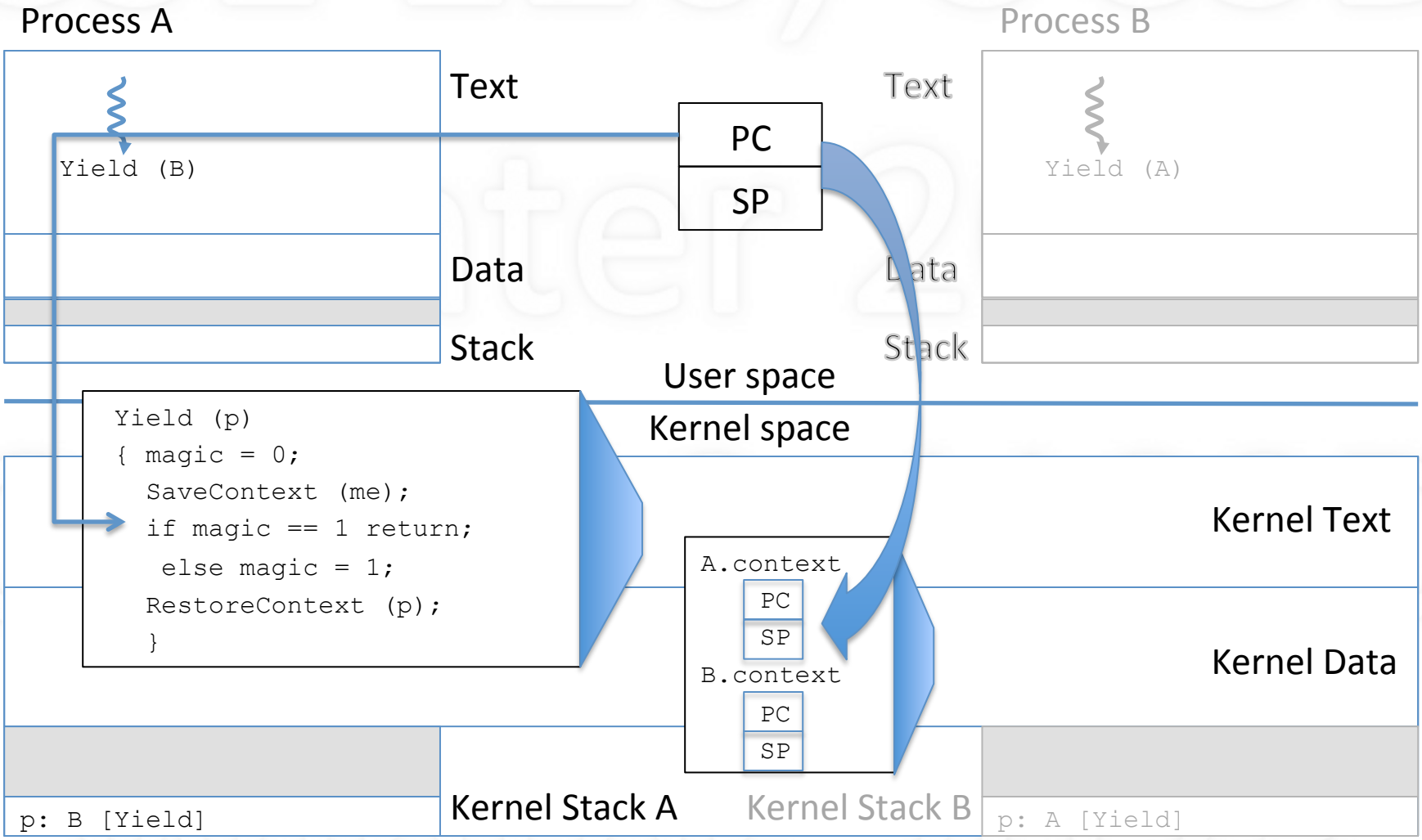
Trap Causes Kernel Entry



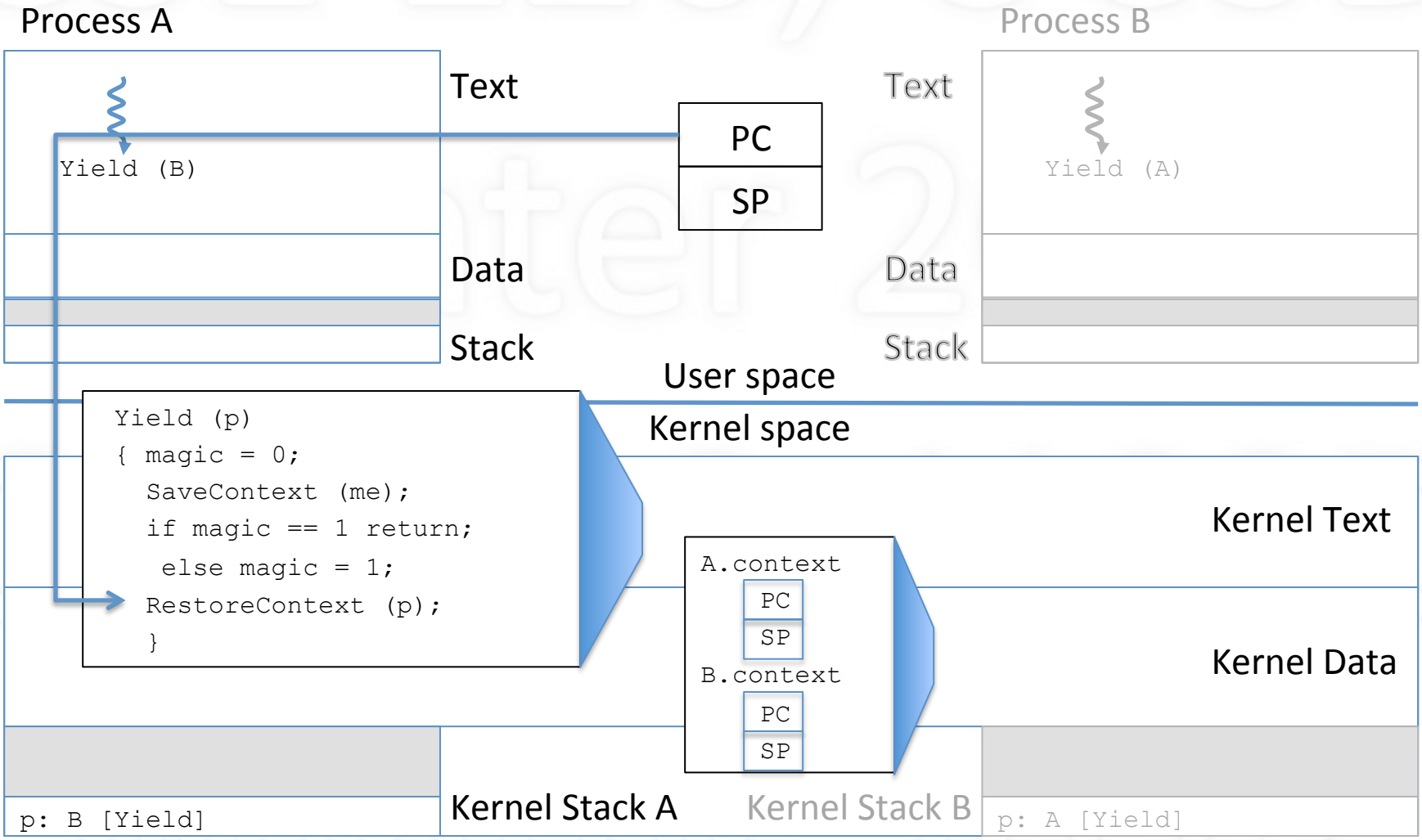
Routine for Yield Executes



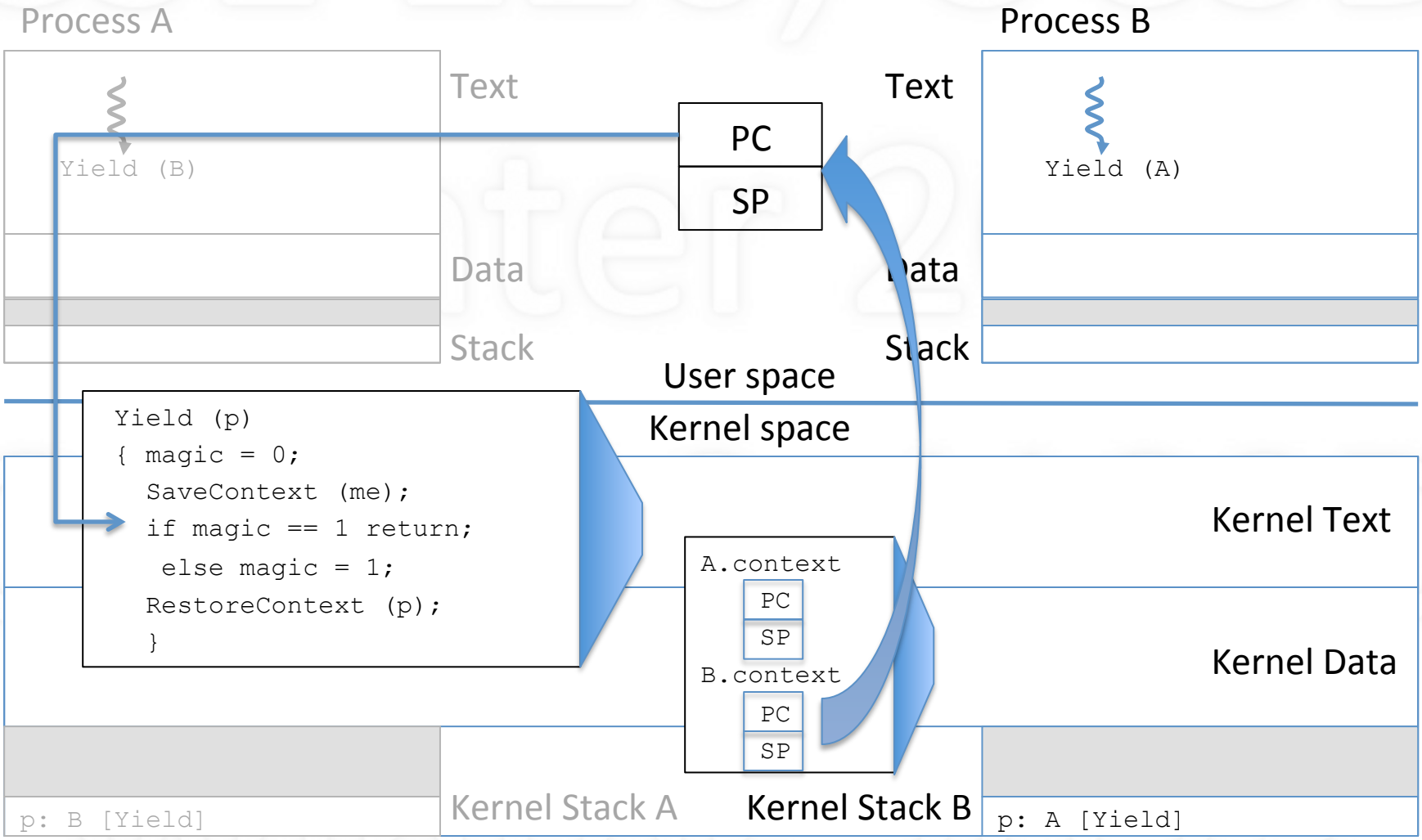
After Saving Context of A



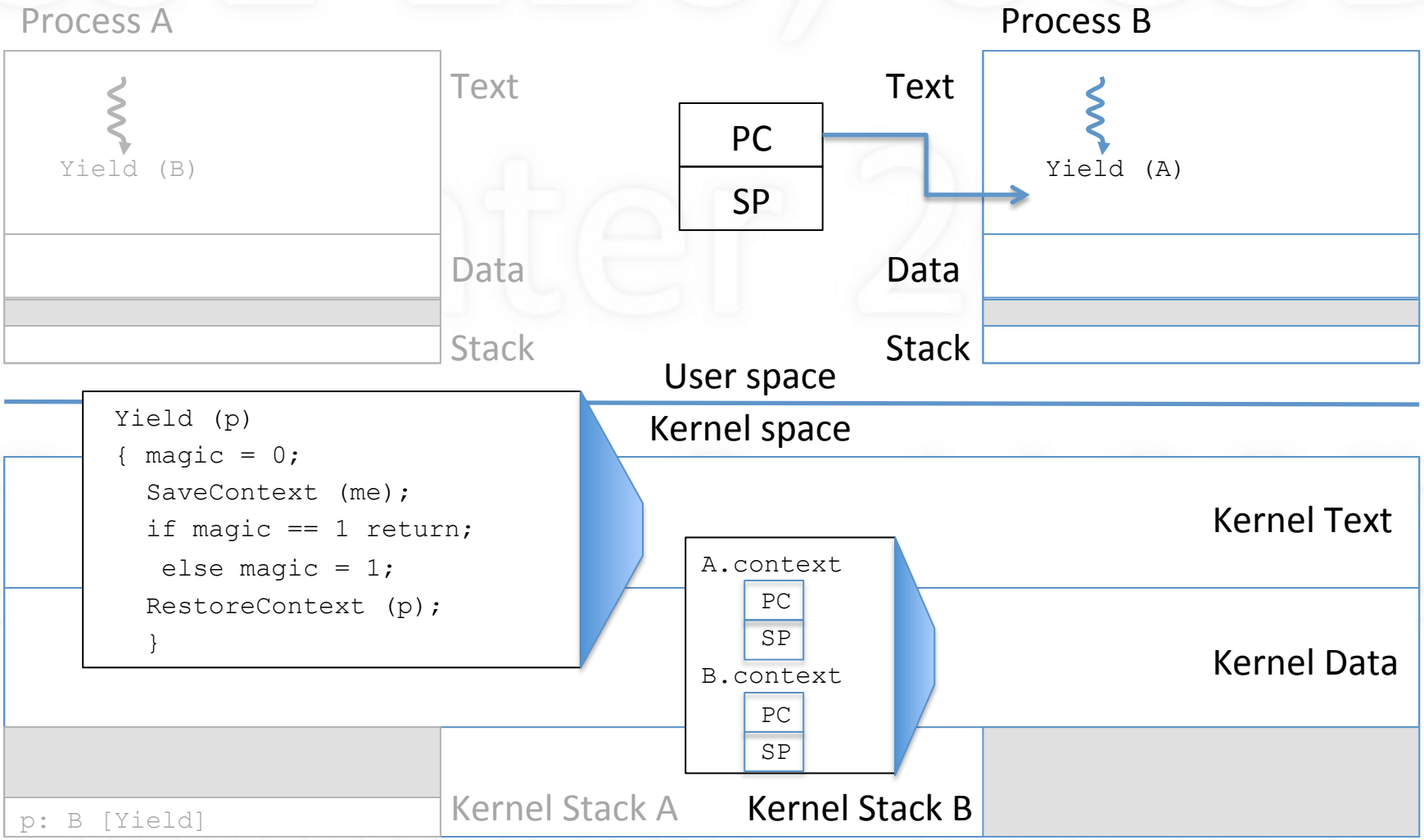
About to Restore Context of B



After Restoring Context of B



Return from Yield in B



Summary

- Process
  - abstraction of a running program
- Multiprogramming
  - allow for multiple processes (despite single CPU)
- Yield
  - when one process gives up CPU to another
- Context switching
  - mechanism that reassigns CPU between processes

Textbook

- OSP: Chapter 2
- OSC: Chapters 3, 4
  - Lecture-related: 3.1-3.3, 3.9, 4.1, 4.3, 4.8
  - Recommended: 4.2, 4.4-4.7