CSE 120: Principles of Operating Systems
Lecture 2: Processes

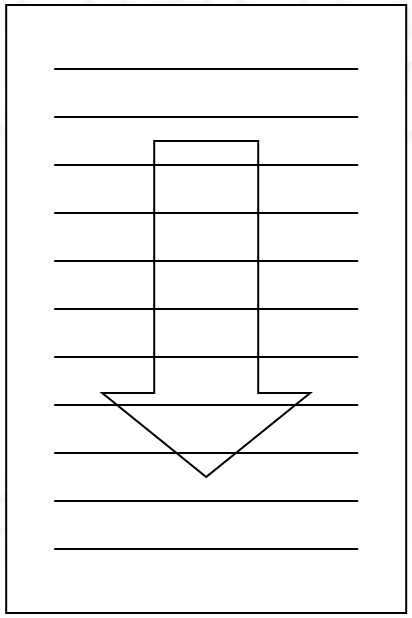
Prof. Joseph Pasquale
University of California, San Diego
January 9, 2019

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

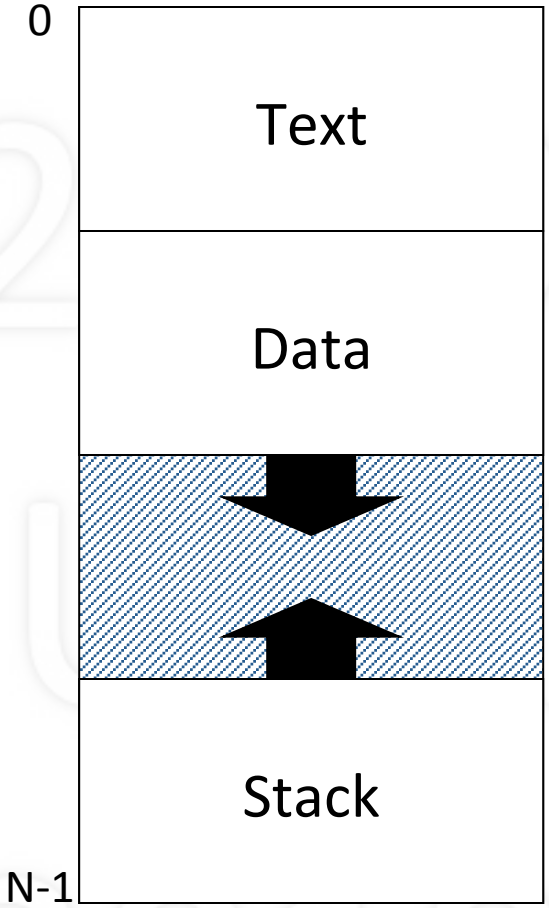
- CPU
  - Processing cycles (time)
  - To execute instructions
- Memory
  - Bytes or words (space)
  - To maintain state
- Other resources (e.g., I/O)

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, …)

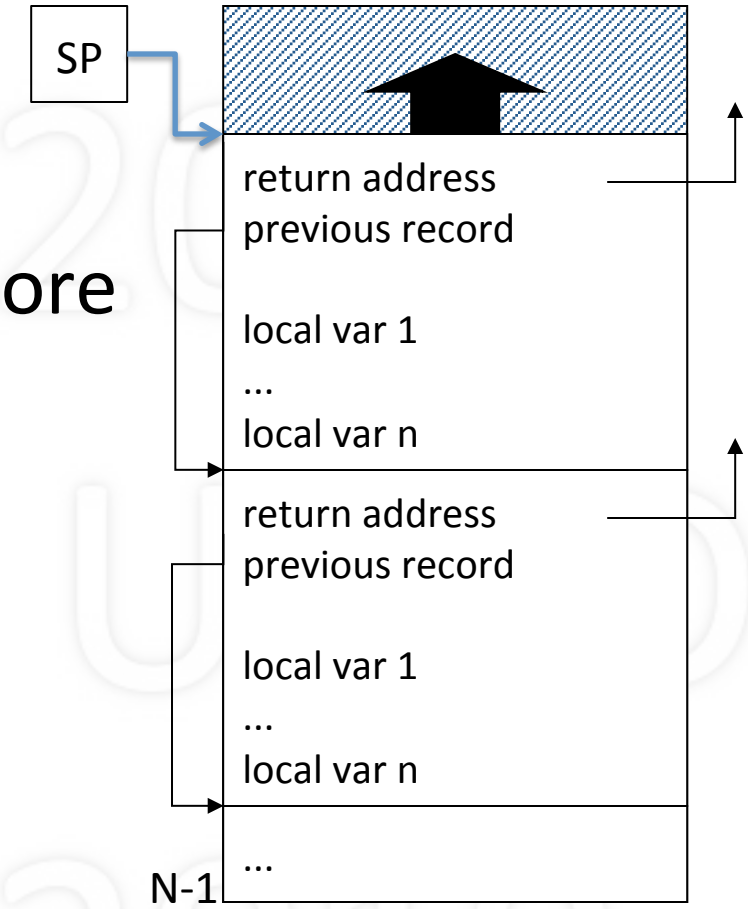
Process Memory Structure

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

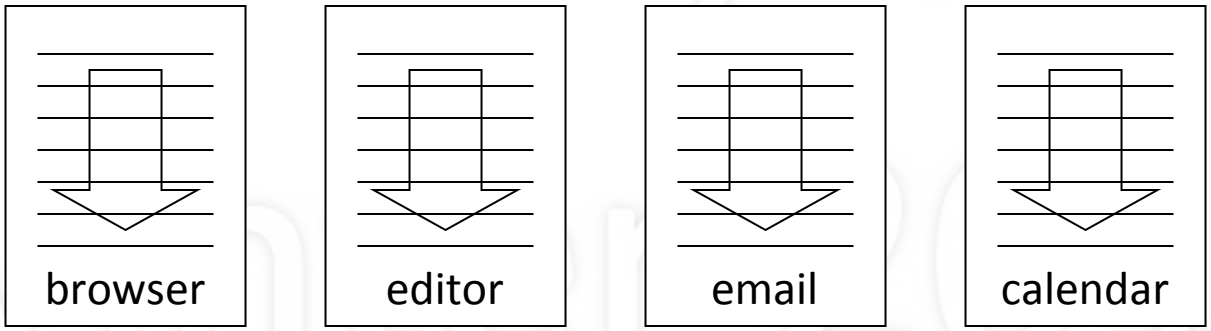


Process Stack

- 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

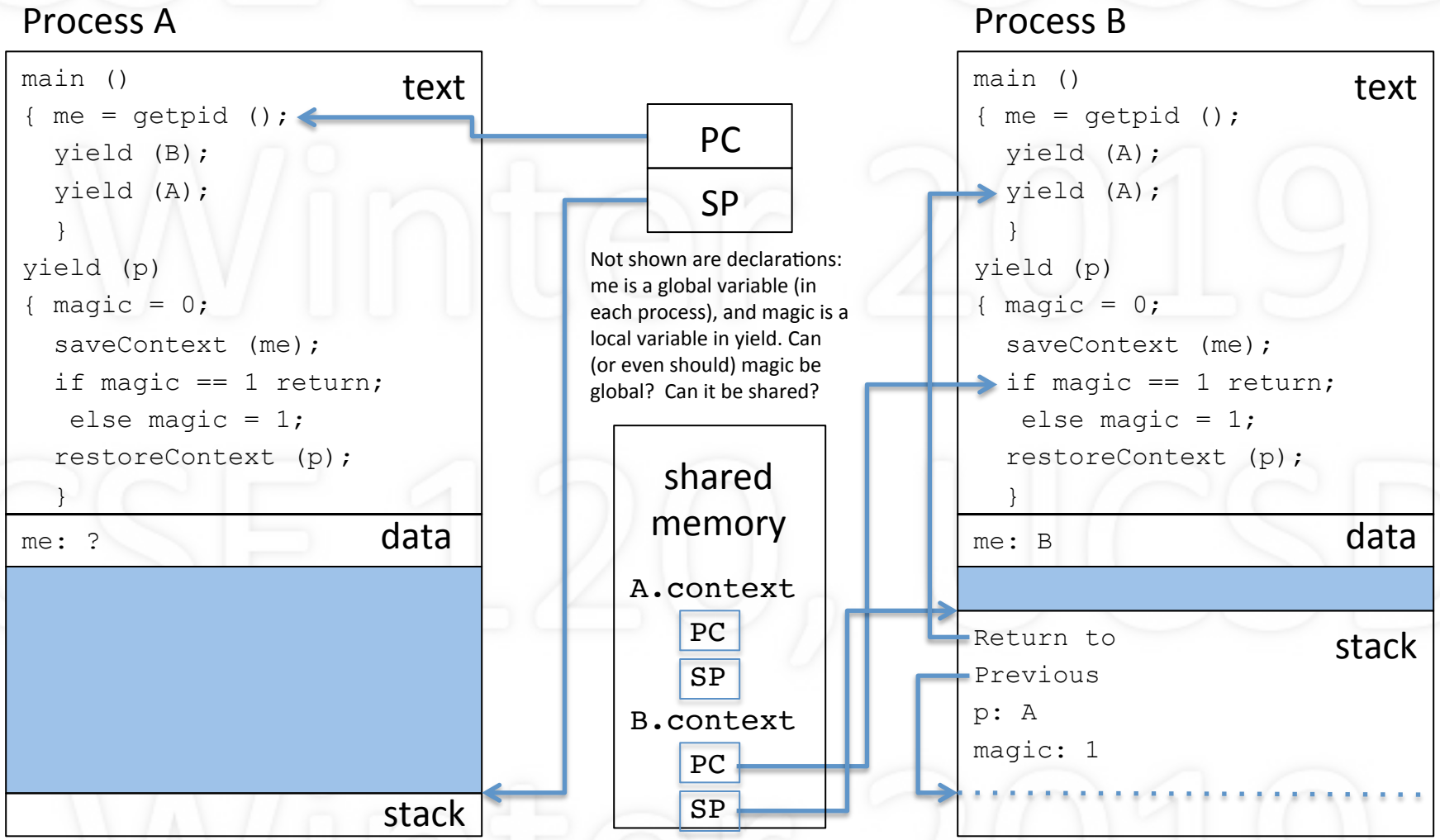
```
magic = 0;                  // local variable
save A's context:           // current process
   asm save GP;             // general purpose registers;
   asm save SP;             // stack pointer
   asm save PC;             // program counter, note value!
if (magic == 1) return;
else magic = 1;
restore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

Example

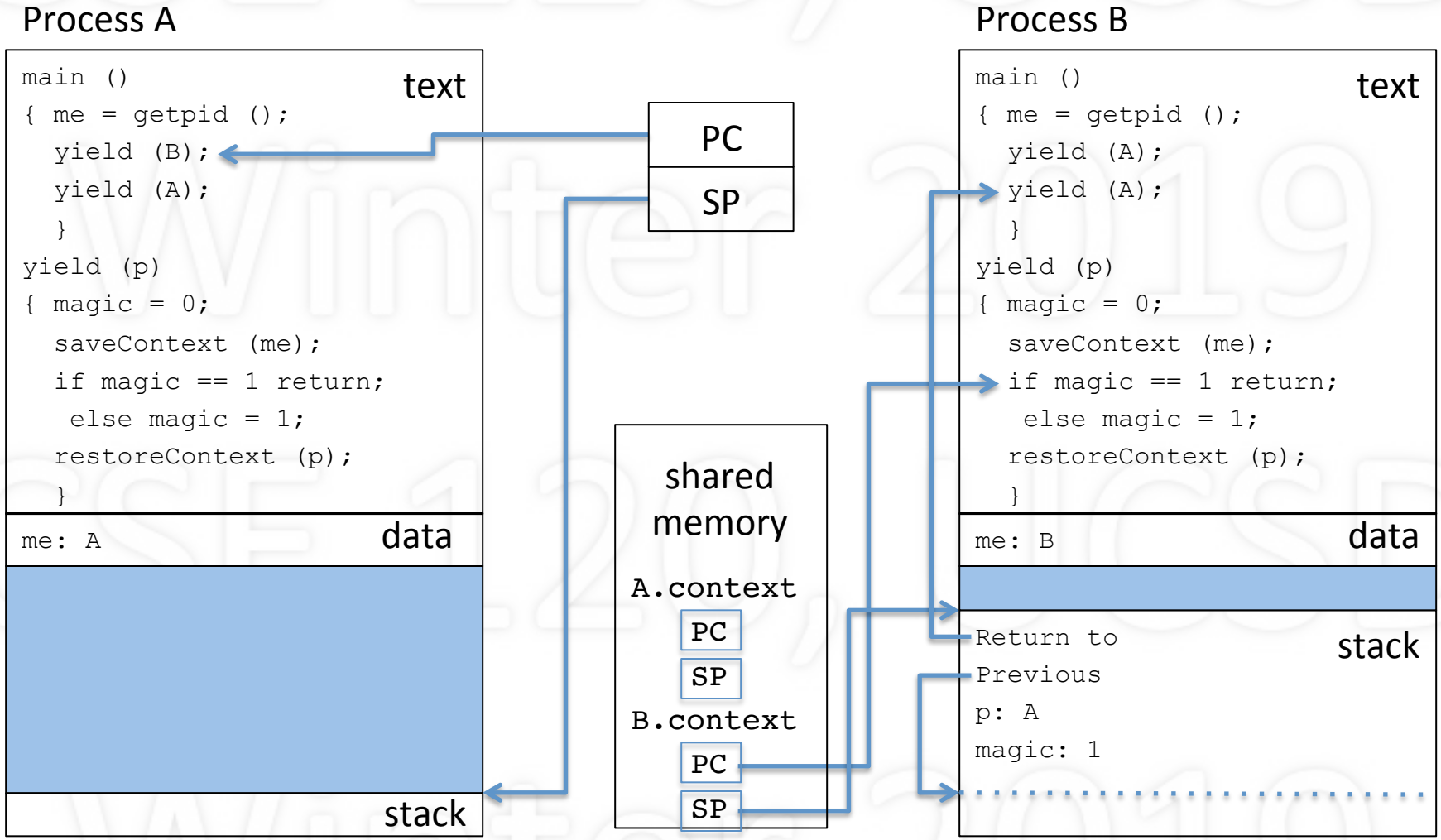
```
int me;

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);
}
```

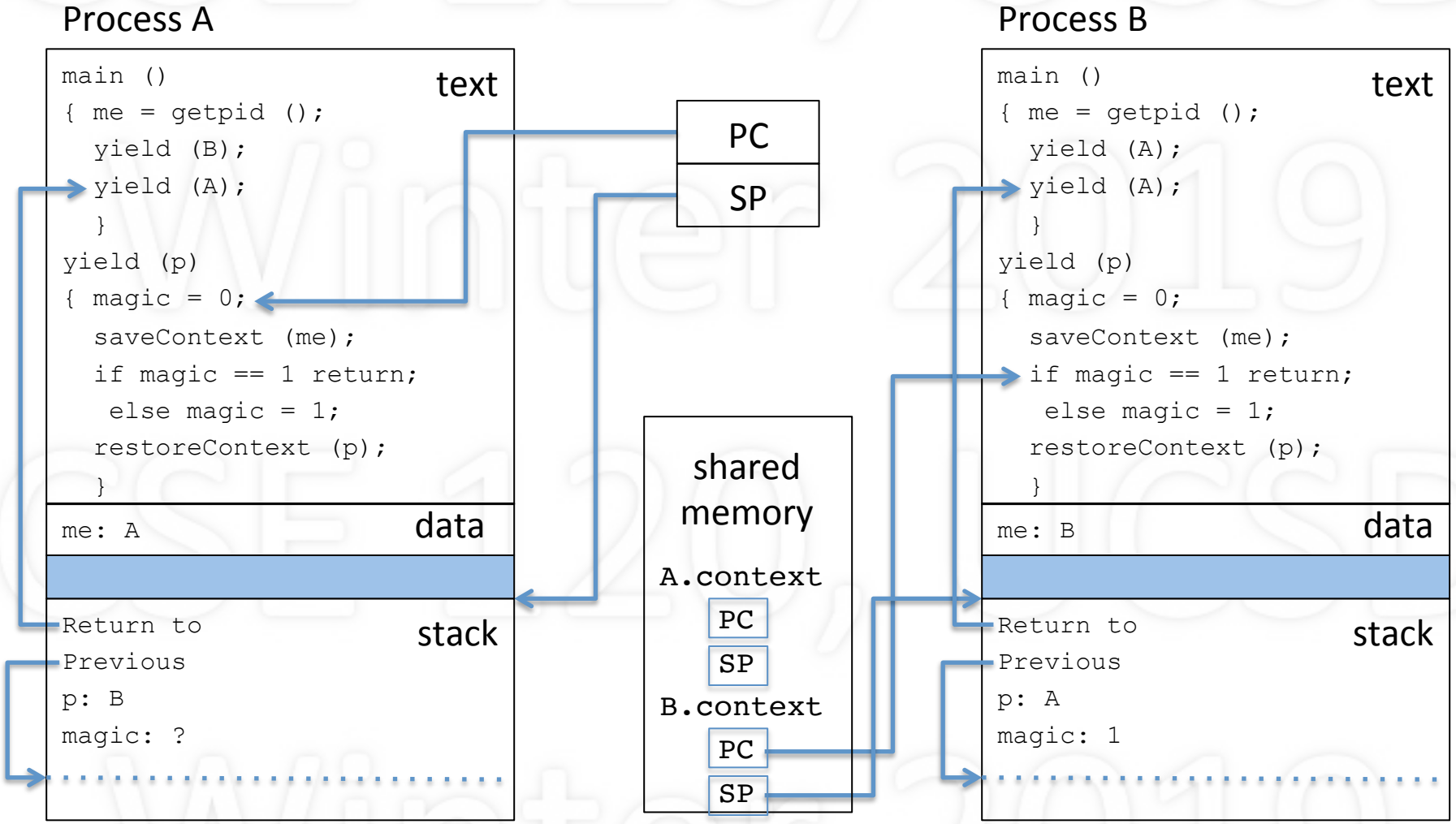
In this example, A is about to set me to its process ID, and yield to B. B had already yielded to A: note B's saved PC and SP.



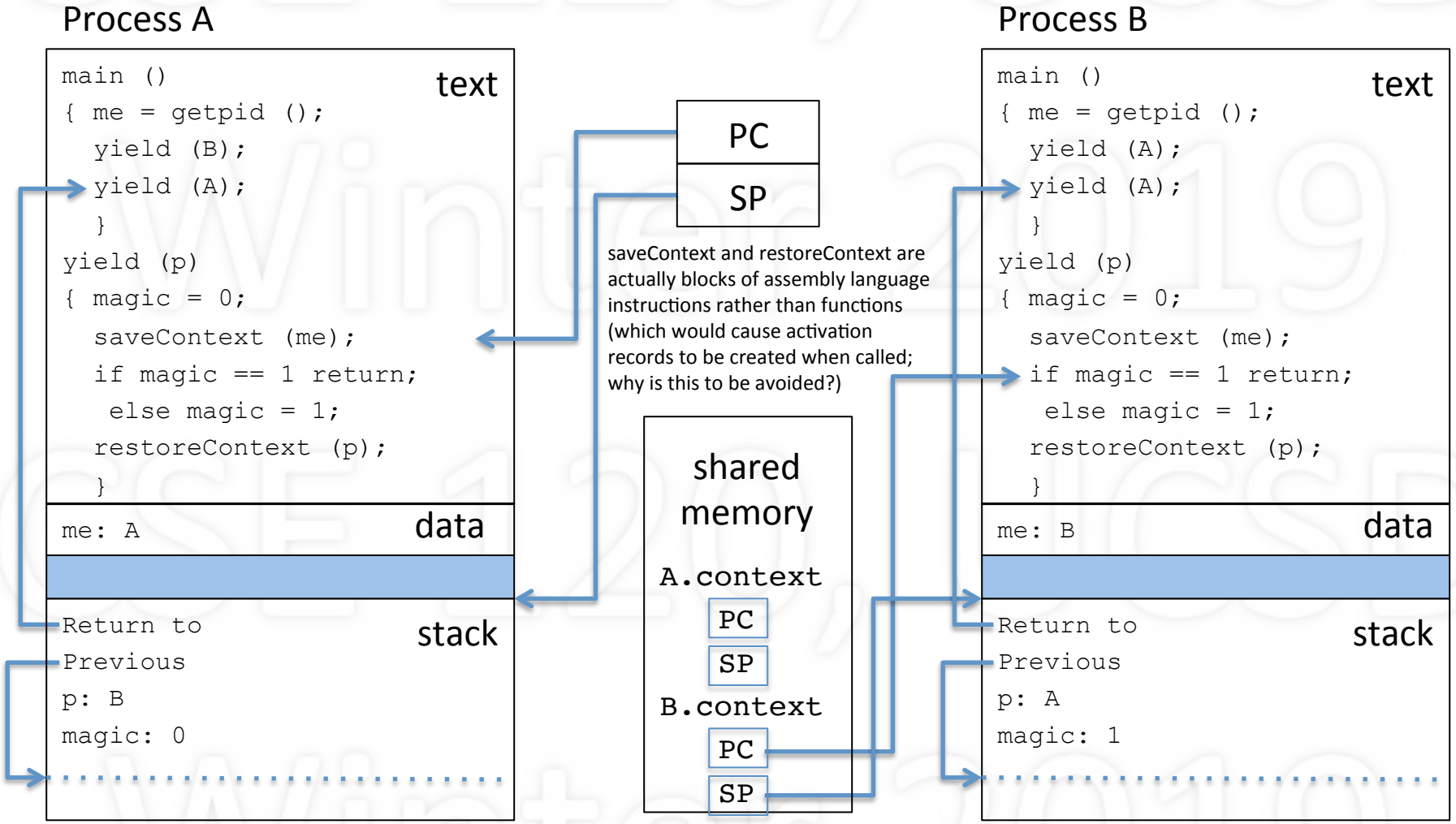
Process A has just set me to ‘A’ and is about to call yield. The PC always points to the next instruction to be executed.



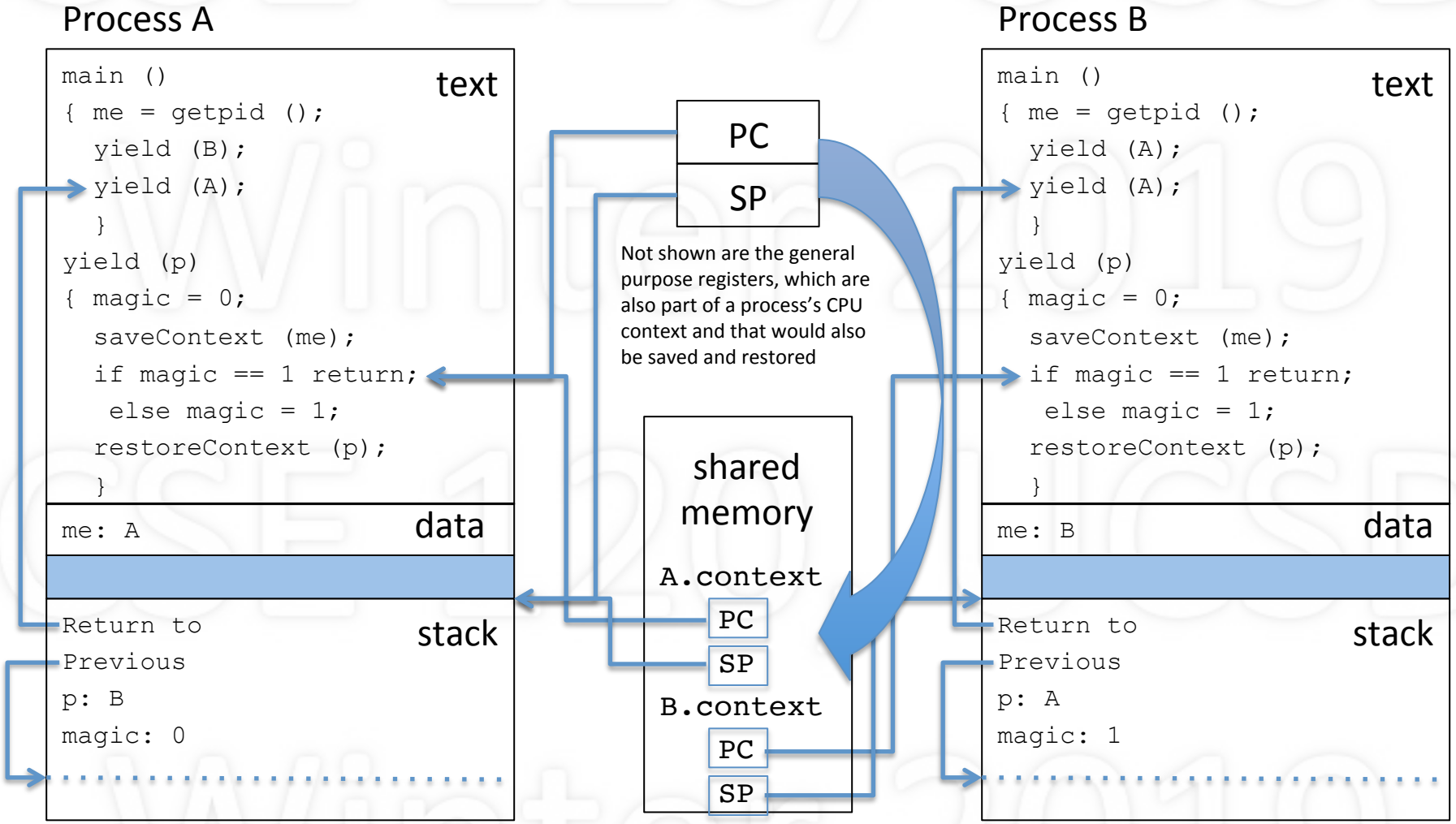
Upon entering yield, an activation record is pushed on the stack. It contains links, and local variables p and magic.



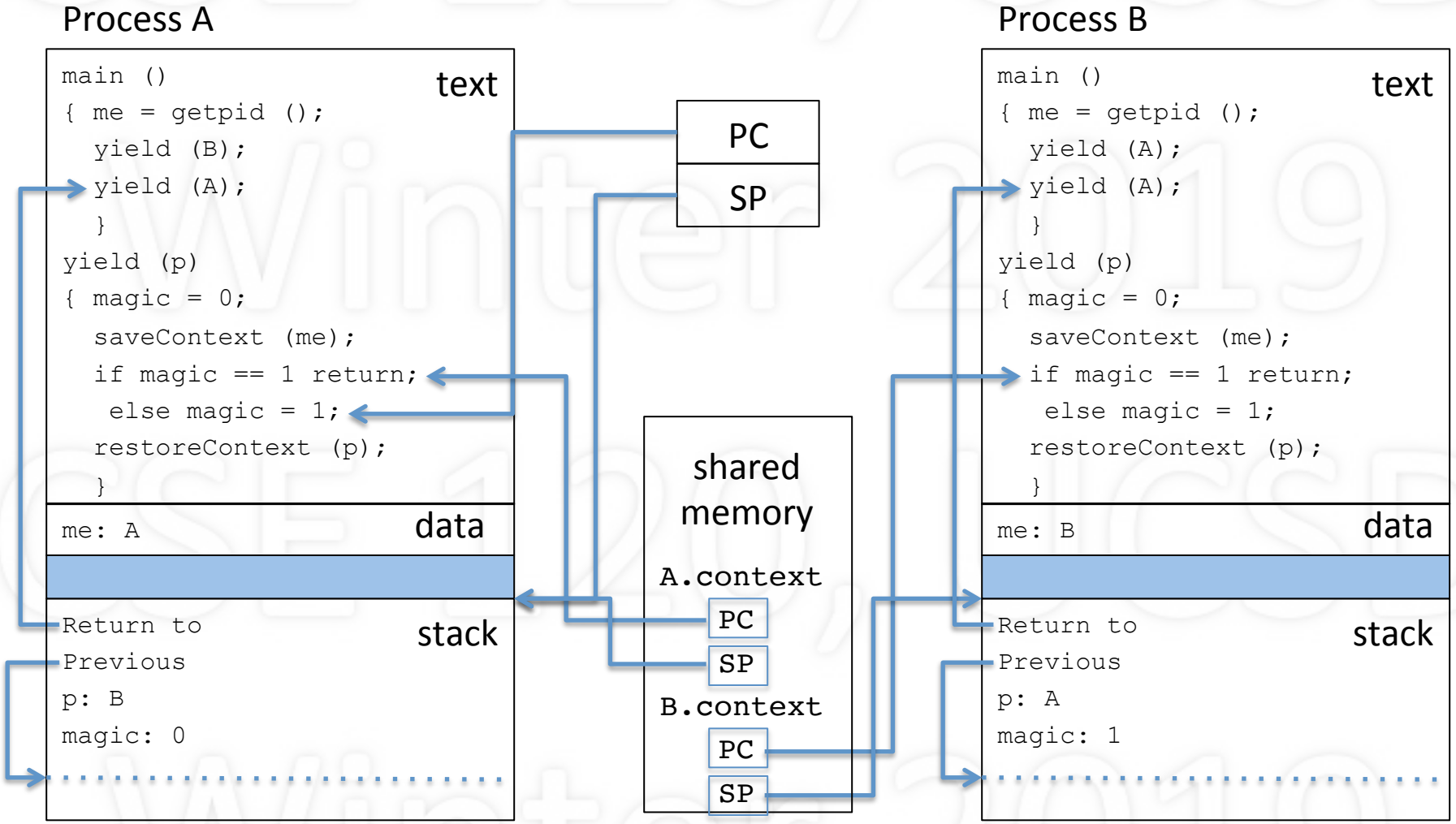
Variable magic is set to 0. It is an automatic variable, dynamically allocated on the stack. Next: save context.



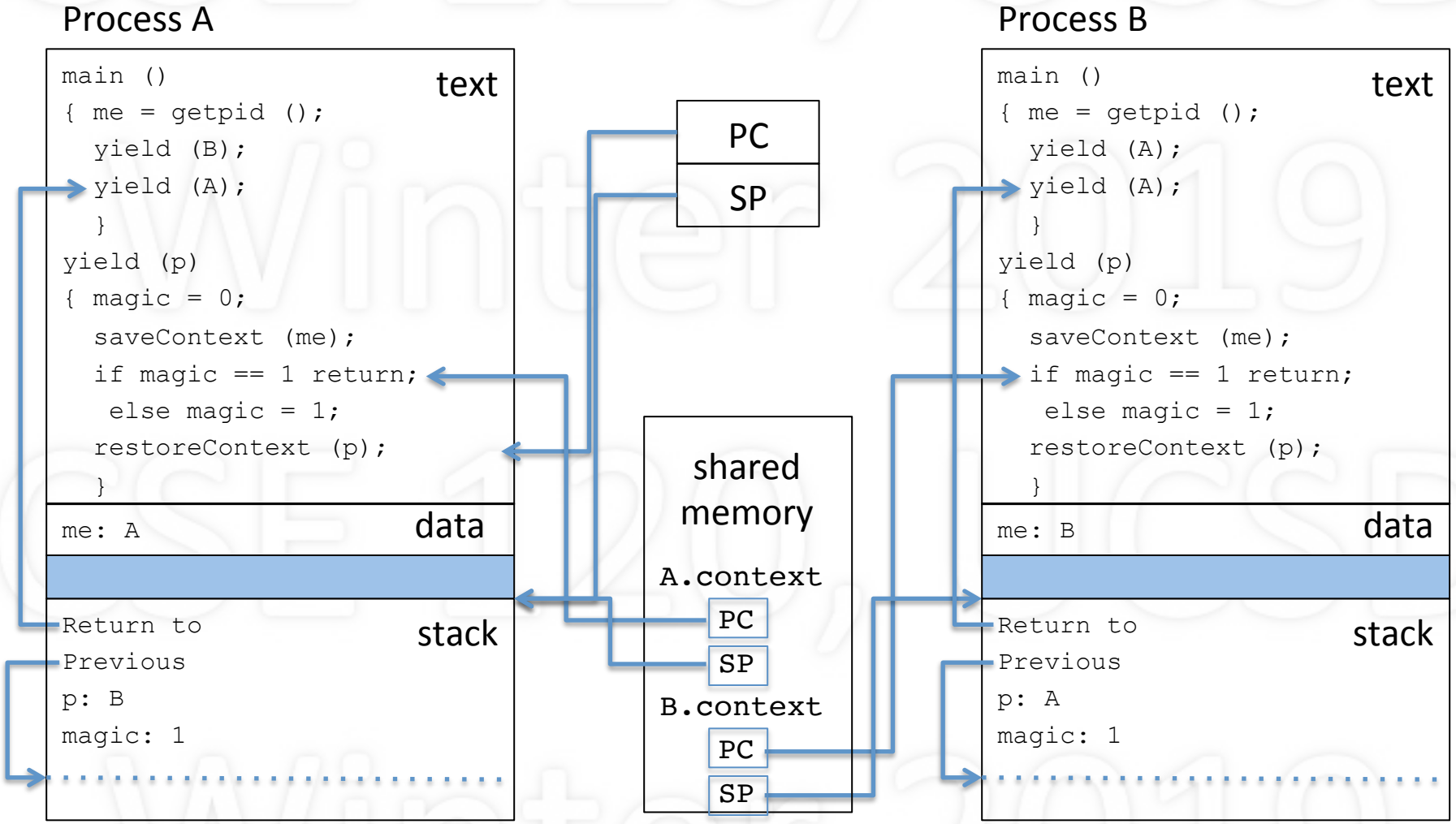
A's context is now saved. The saved PC points to just after the call to saveContext. Compare this to B's saved context.



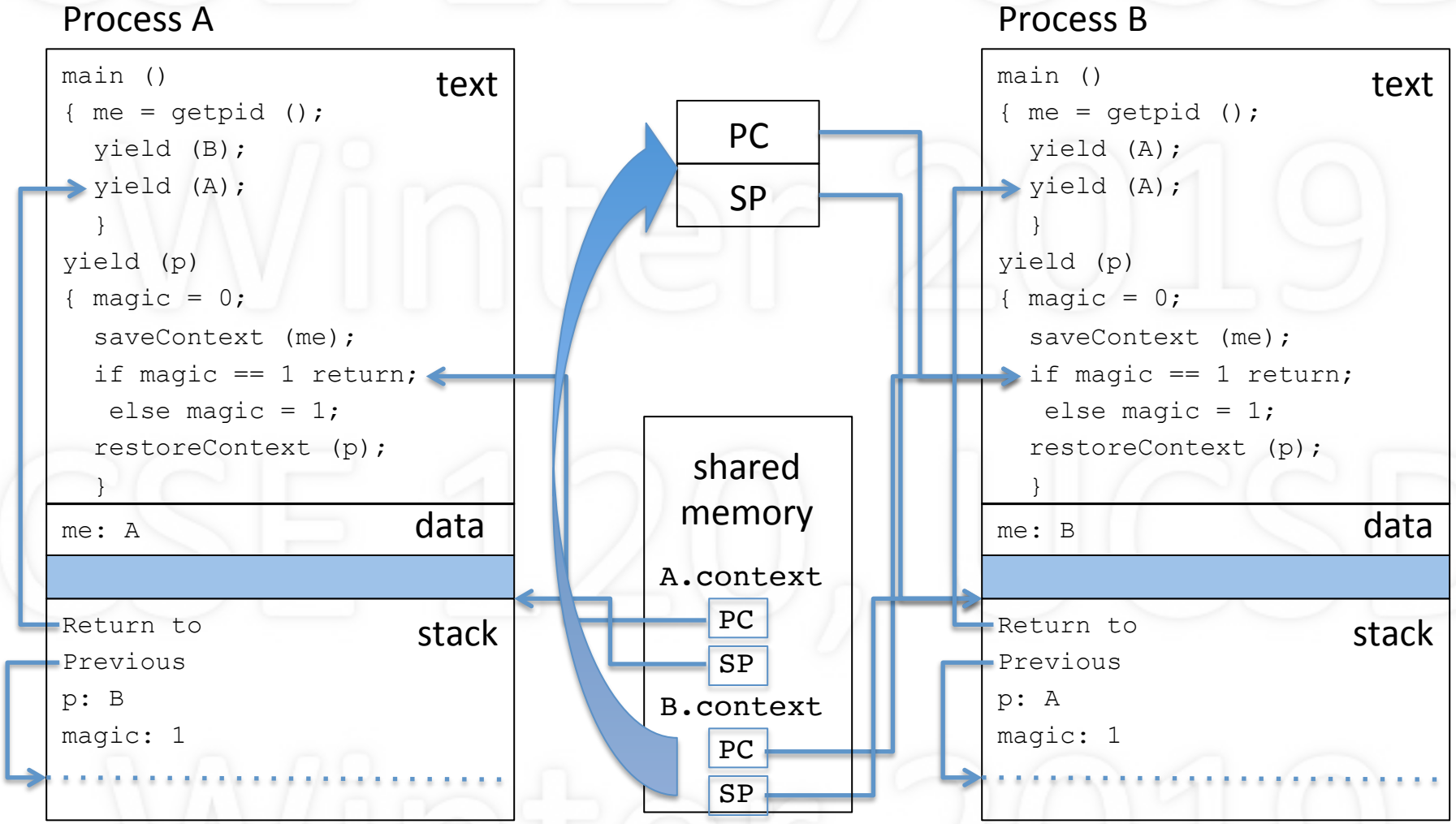
A just checked whether magic equals 1, which was false, and so, on to the else clause to set magic to 1.



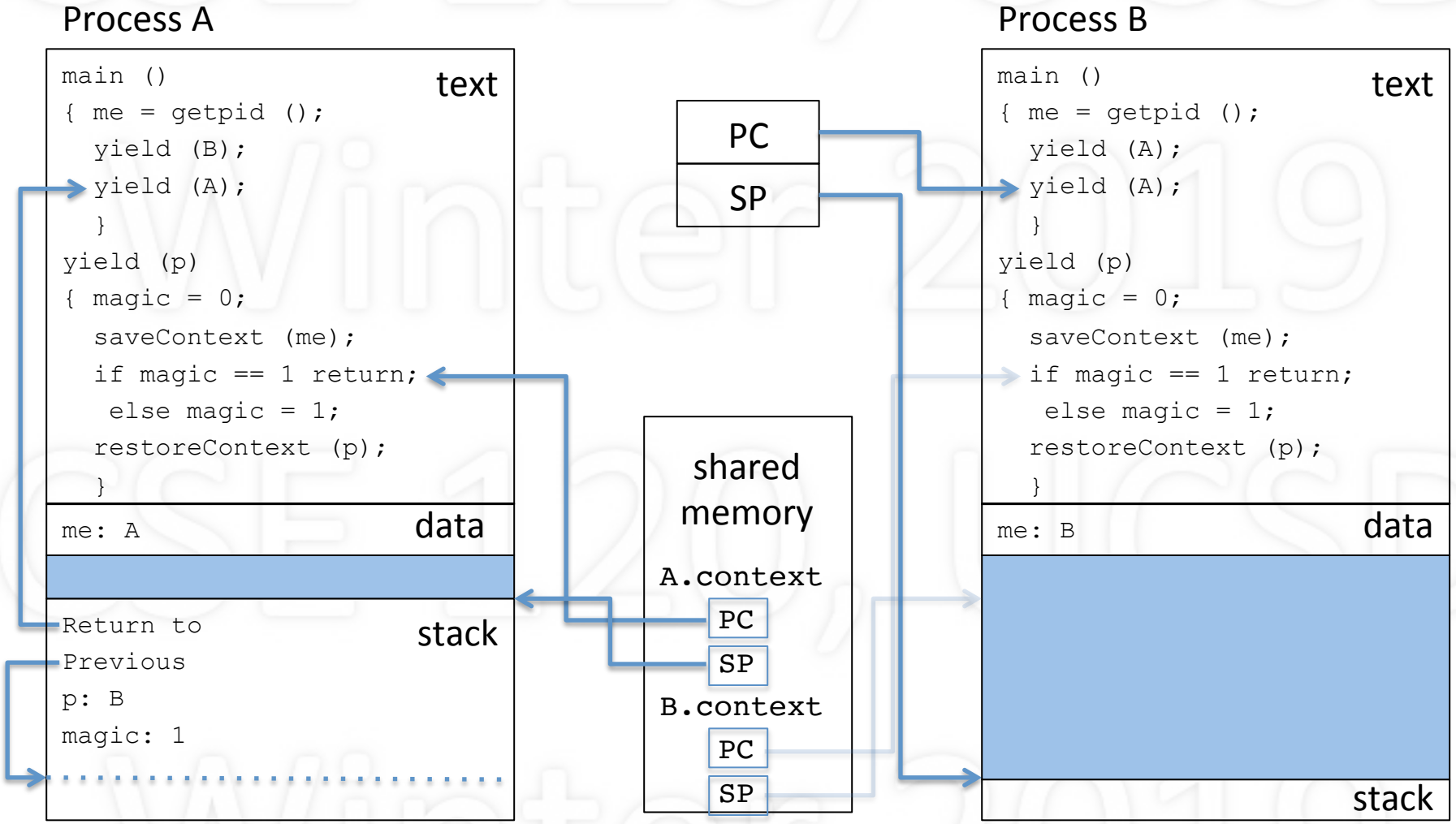
A sets magic to 1, and is about to restore B's context (just like B's situation in the past when it restored A's context).



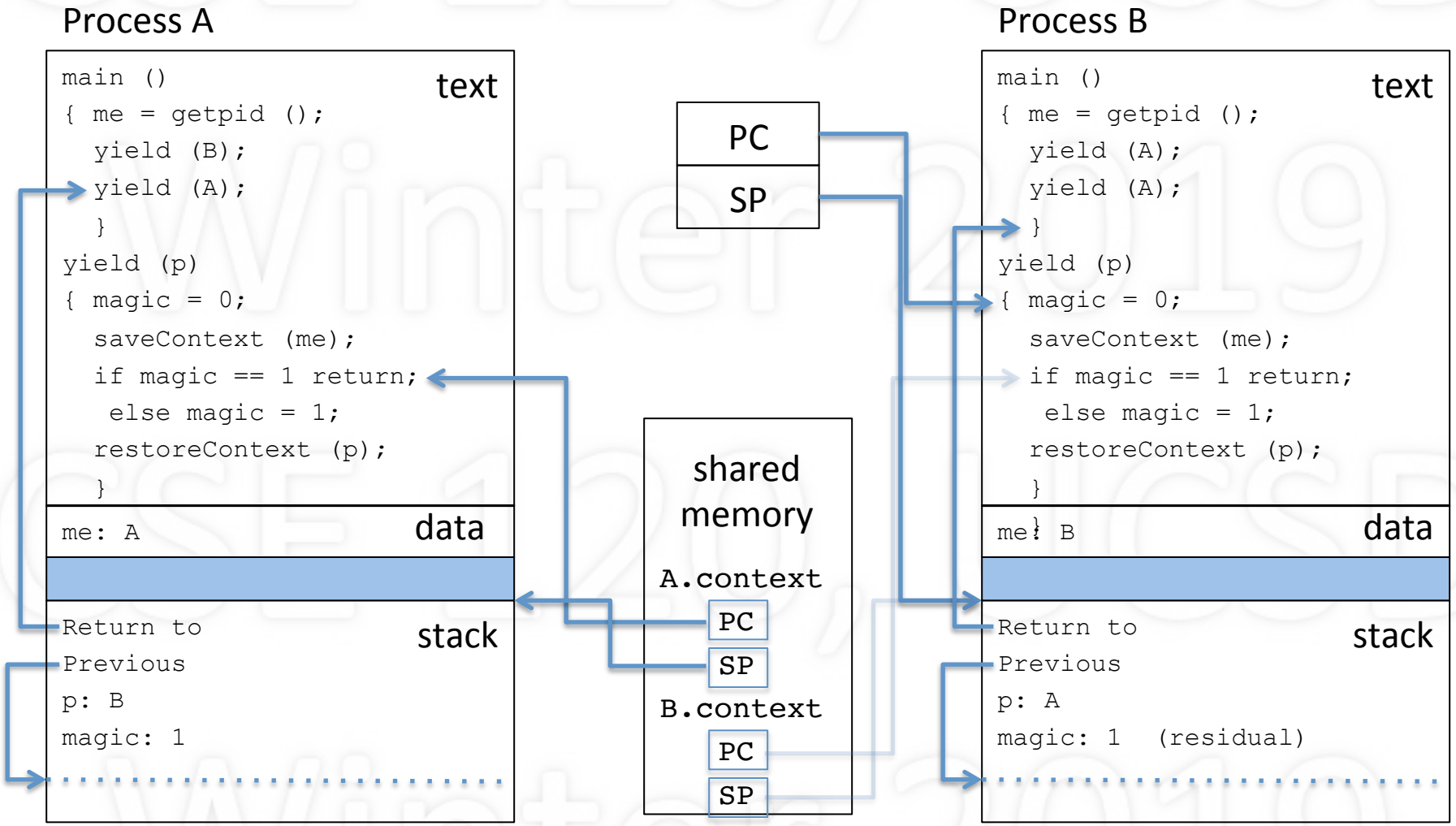
B's context is restored (i.e., the machine state is now that of B). The PC points to the if statement.



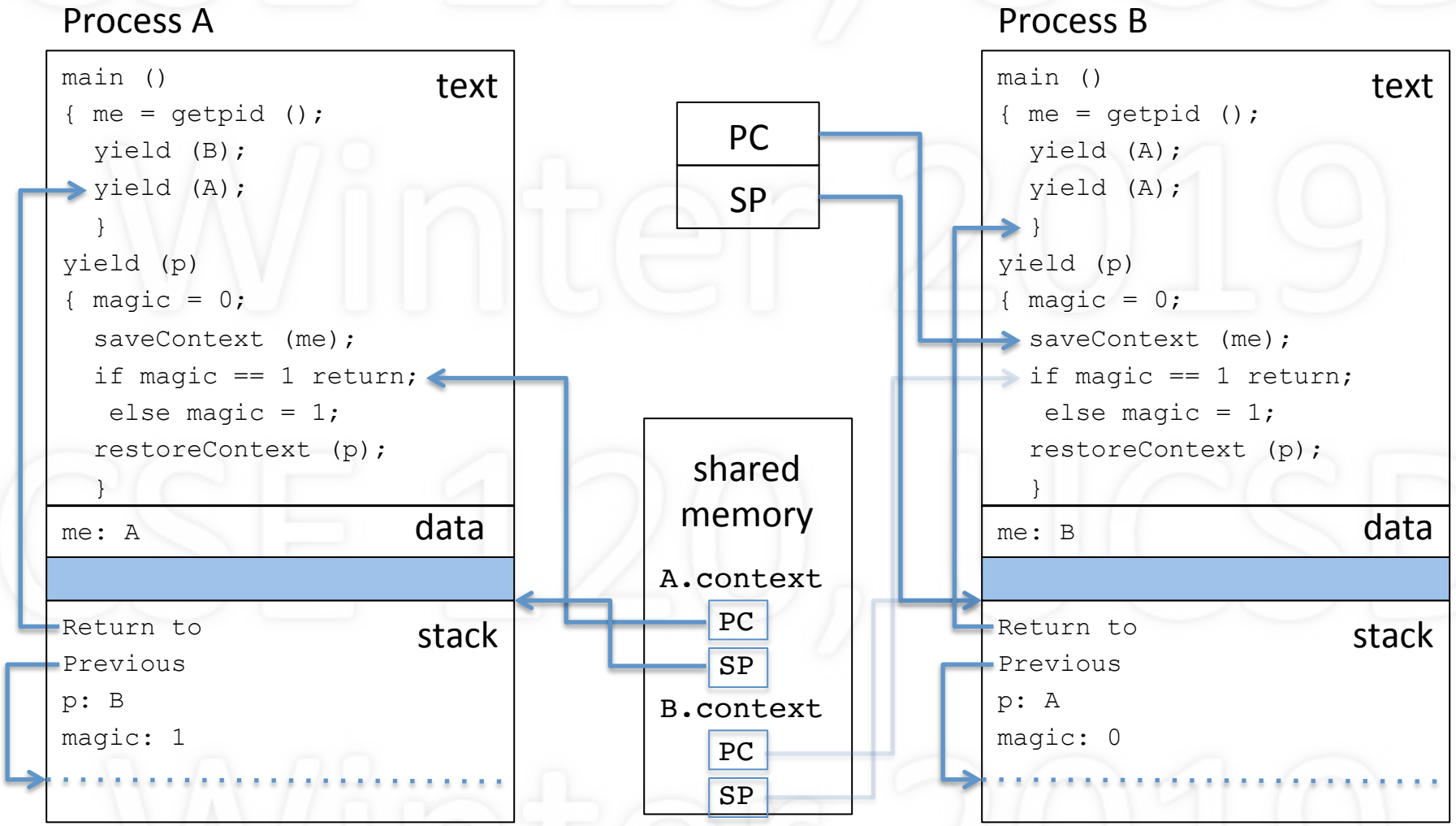
Since magic equals 1, B returns from yield (unlike last time when magic equaled 0). No wonder it's called magic!



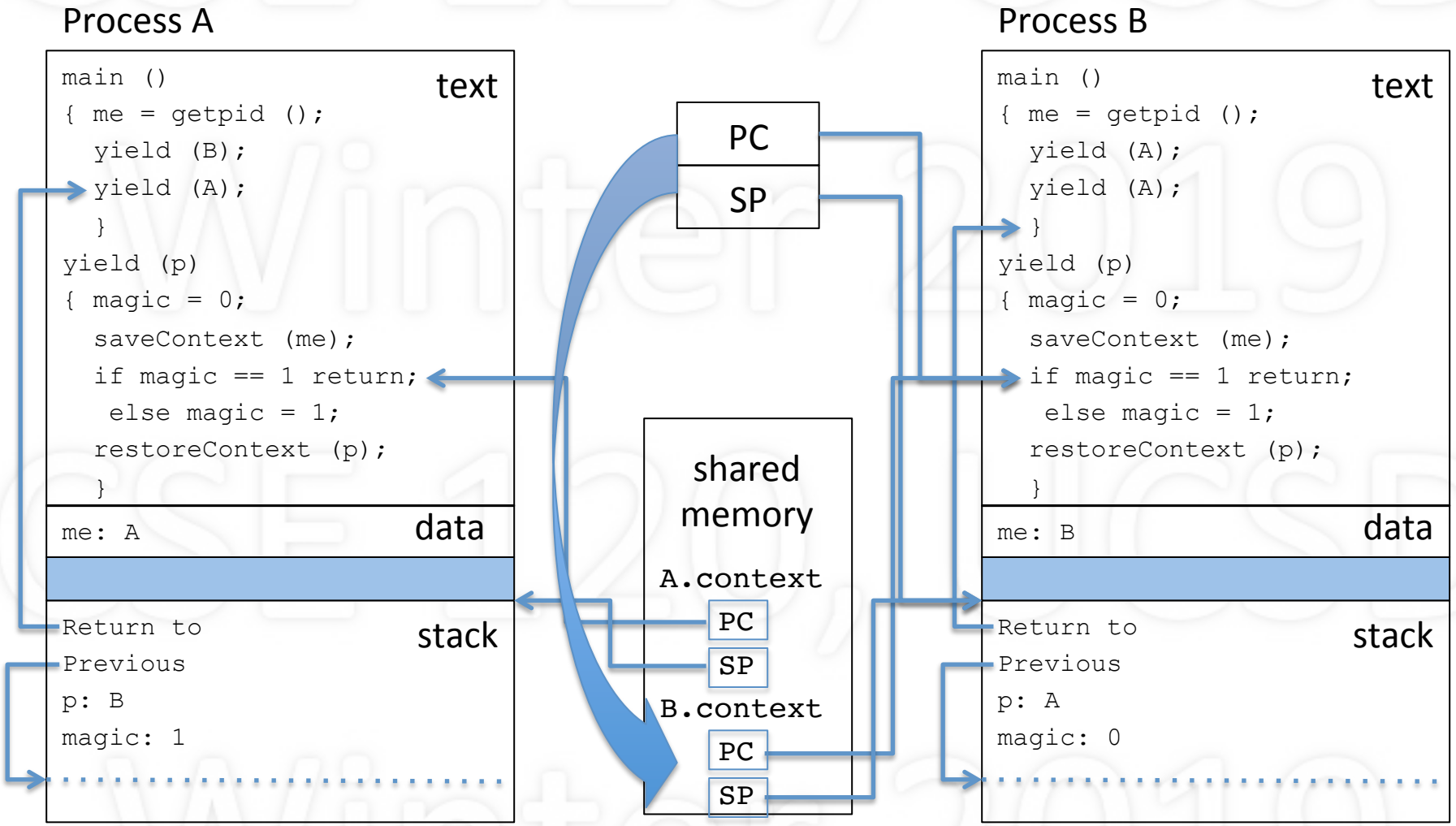
See if you can figure out how the rest of this works.



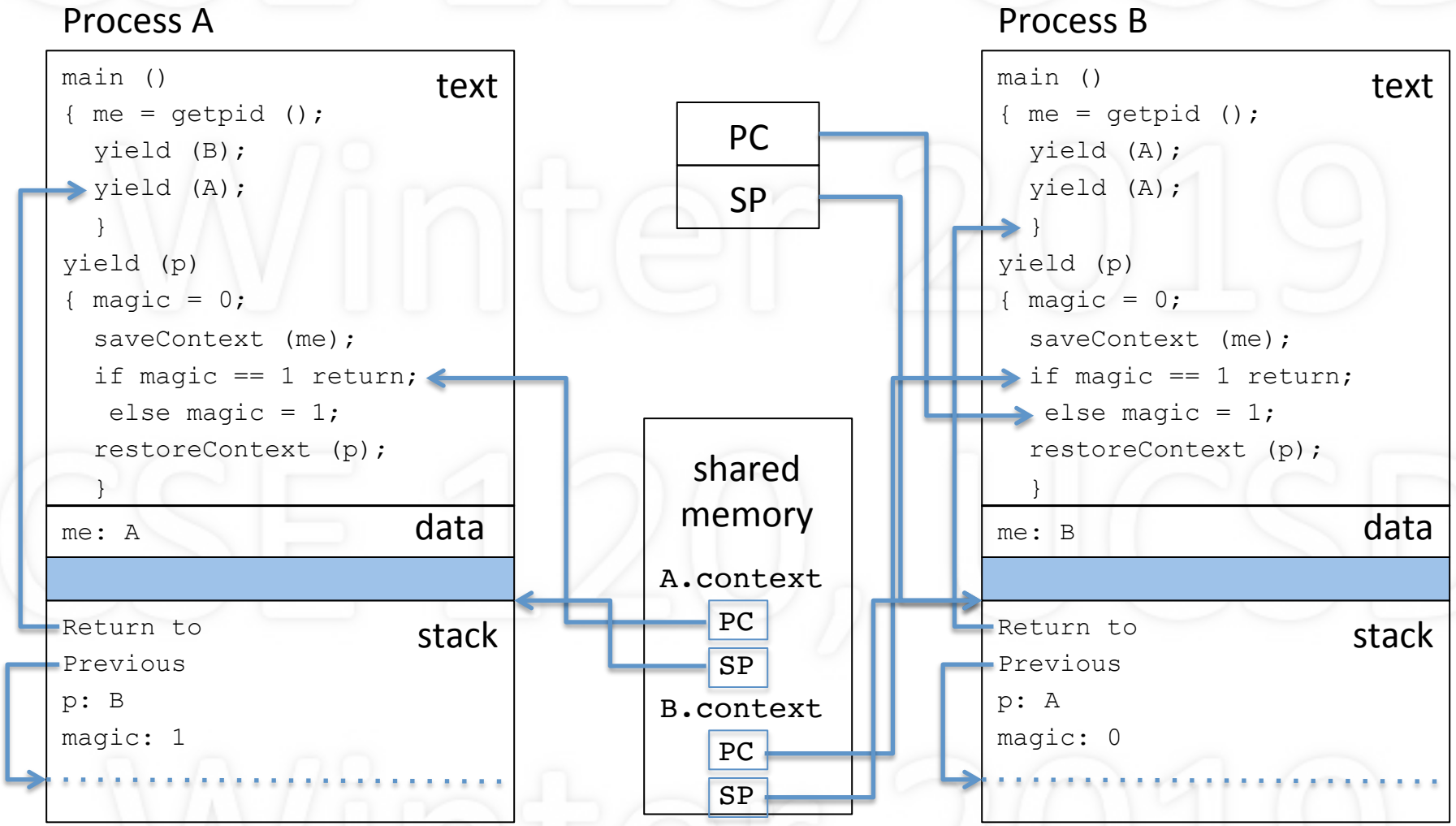
B is yielding to A.



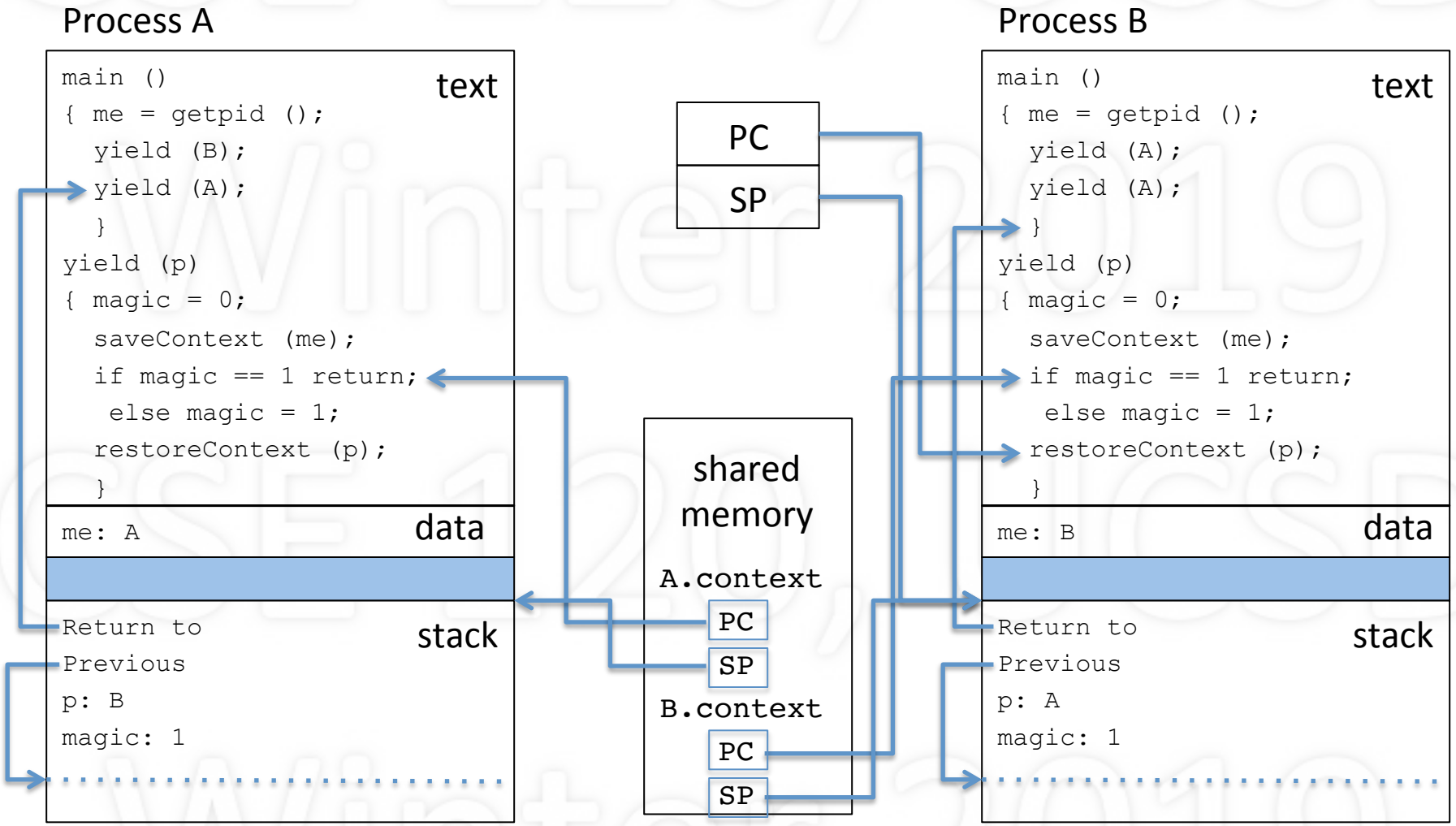
Goes through the same steps as we saw for A.



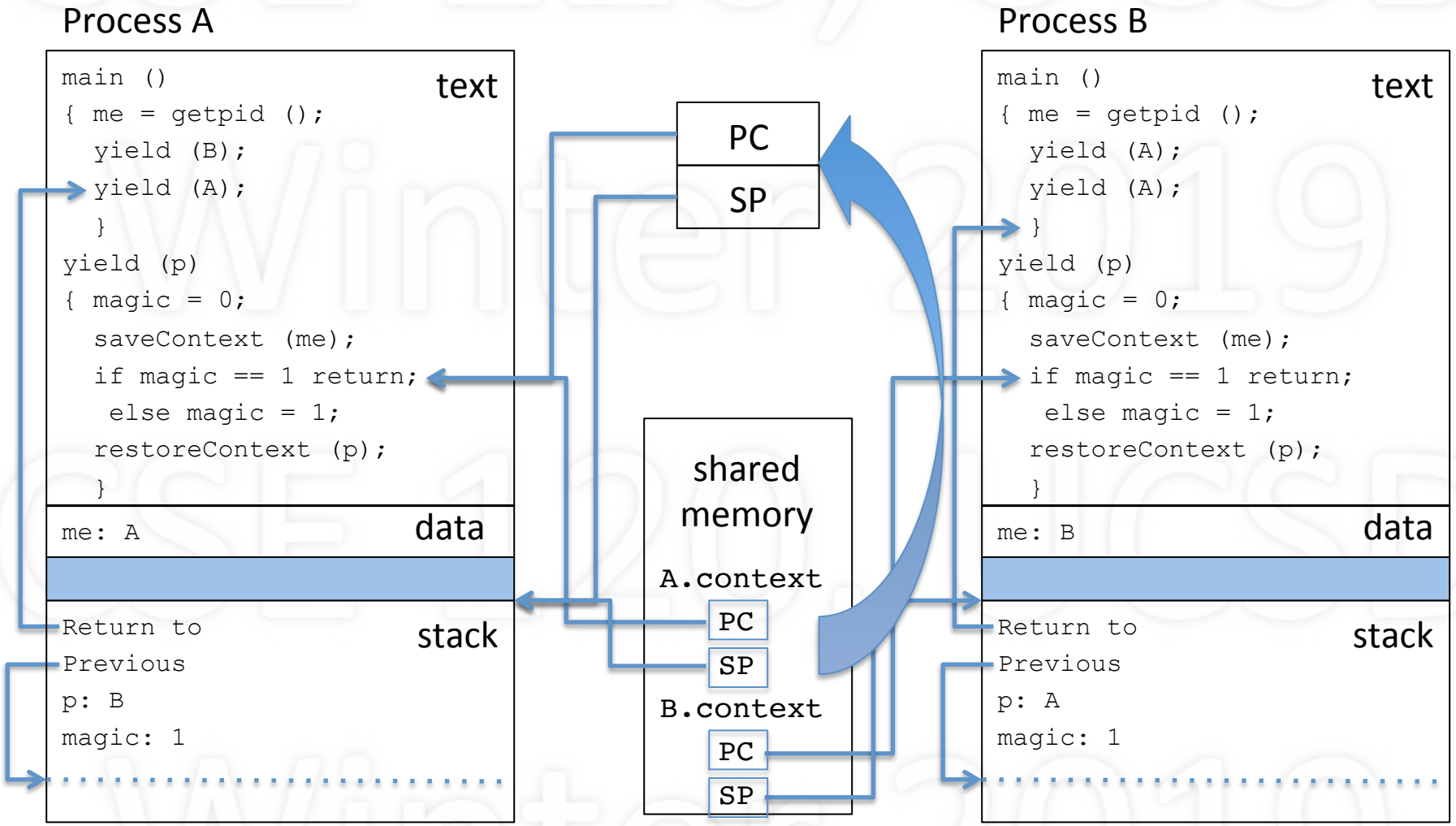
Note value of magic.



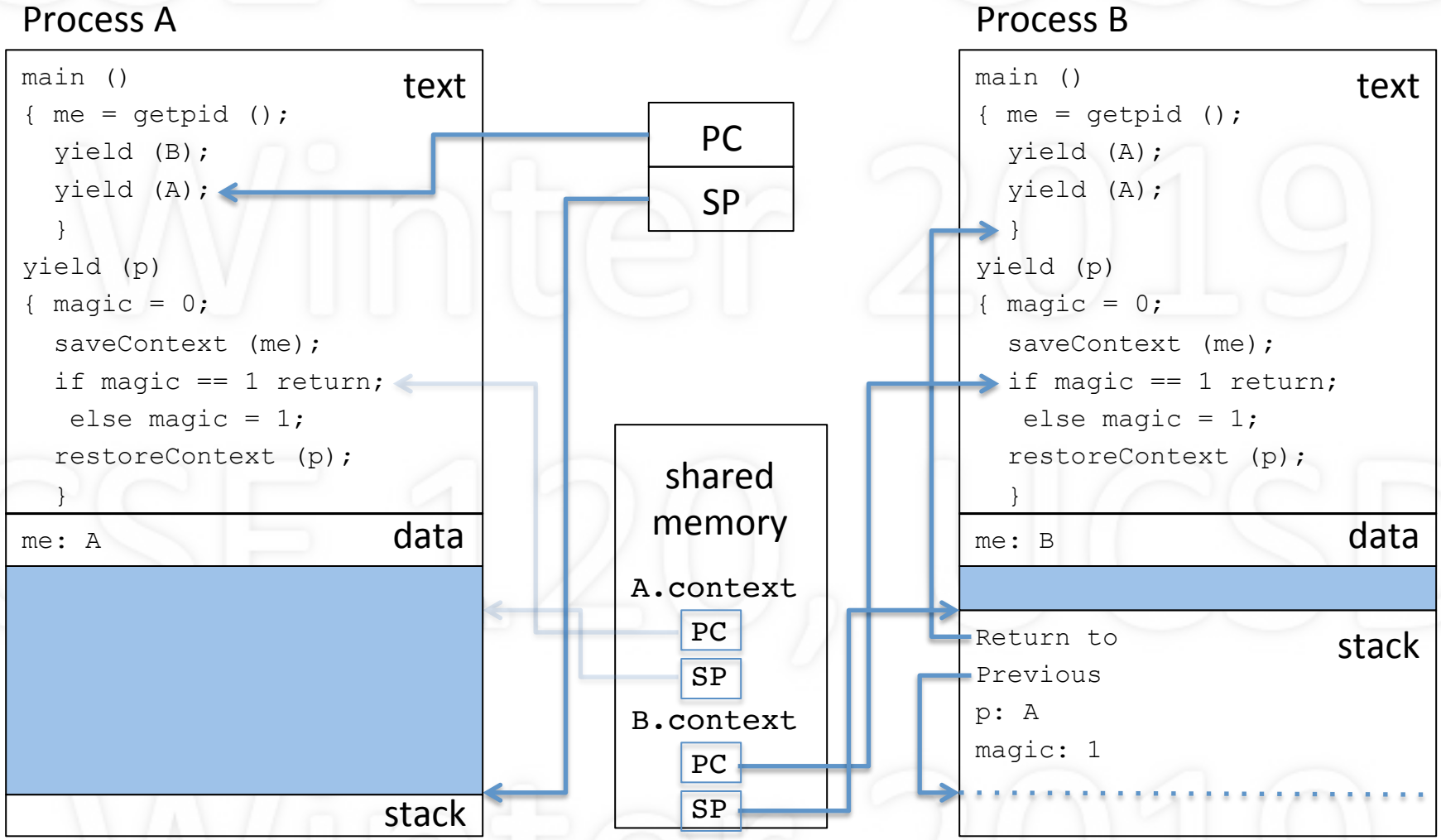
About to restore context of A.



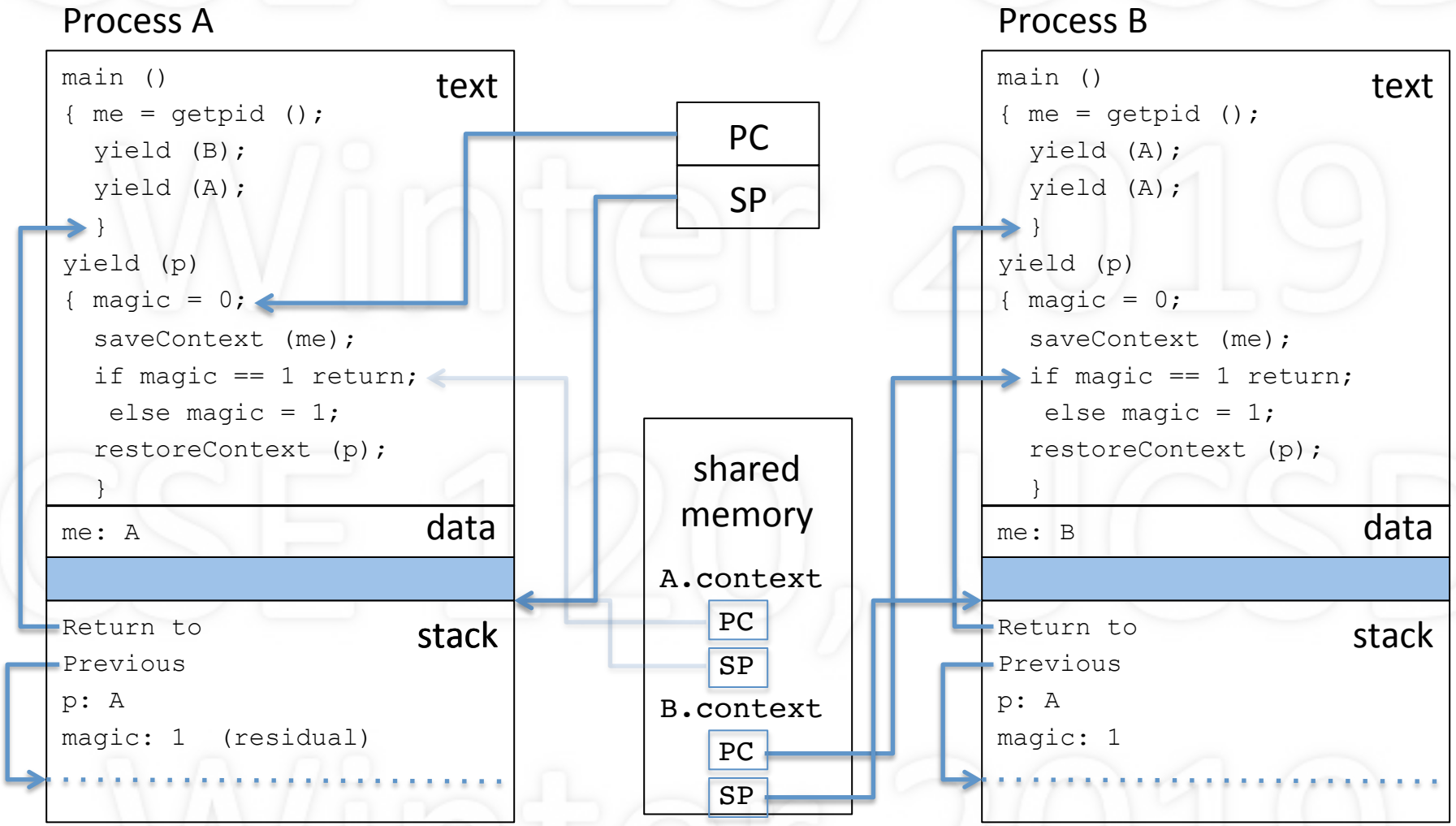
A resumes. Note value of magic.



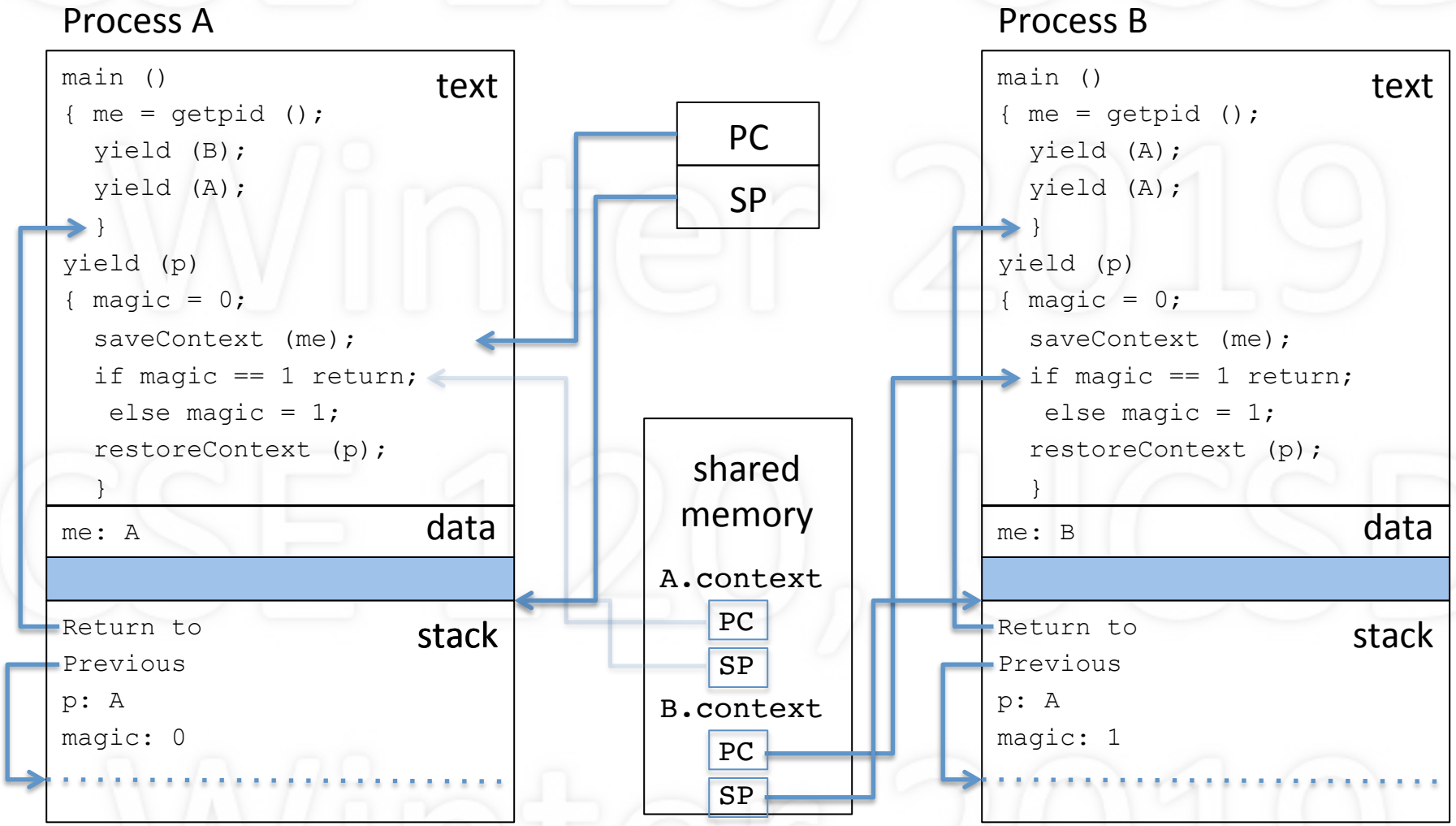
Returns from yield, and now about to call yield again, but to itself!



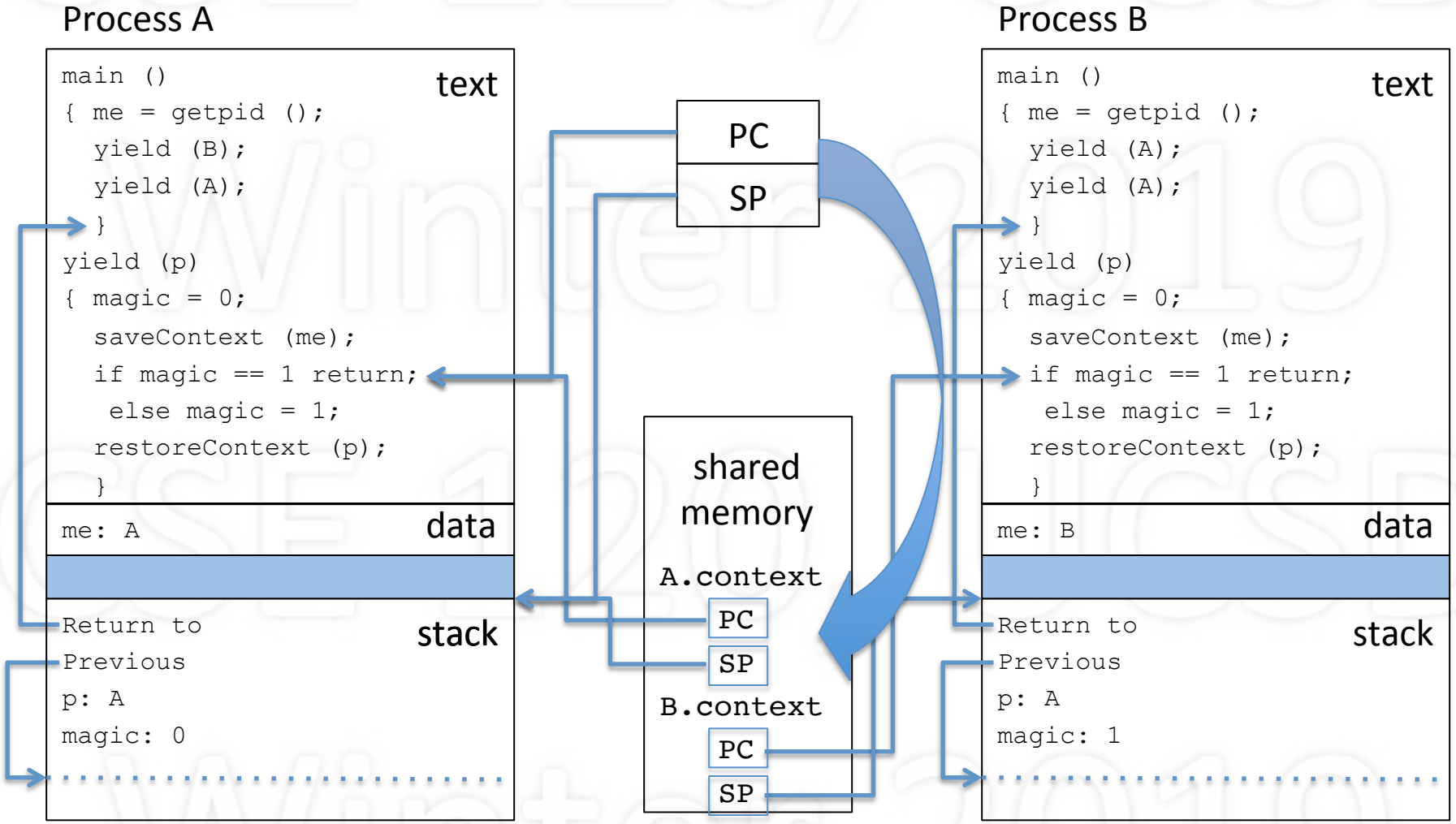
Inside yield. Note new activation record on stack.



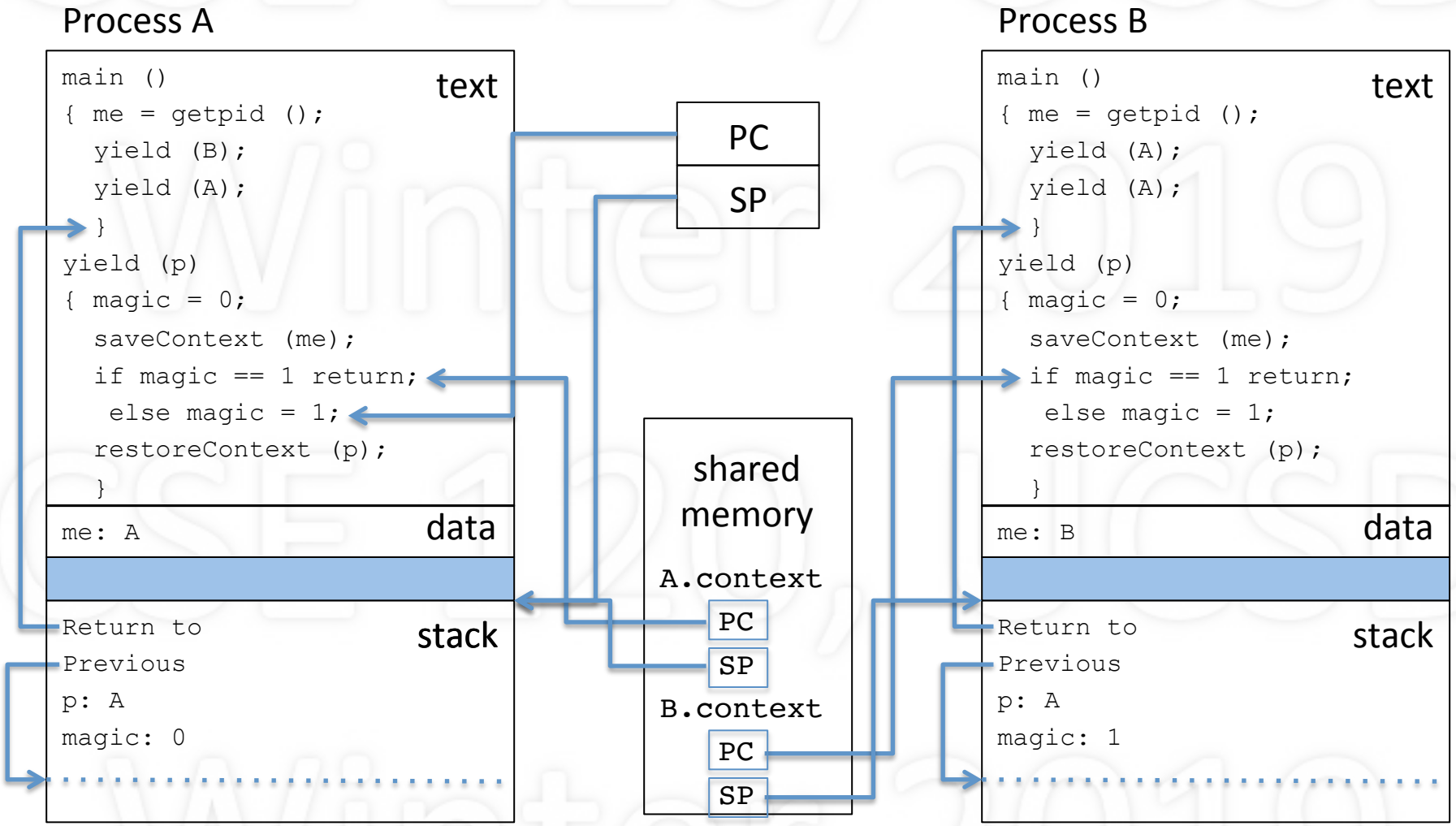
About to save context.



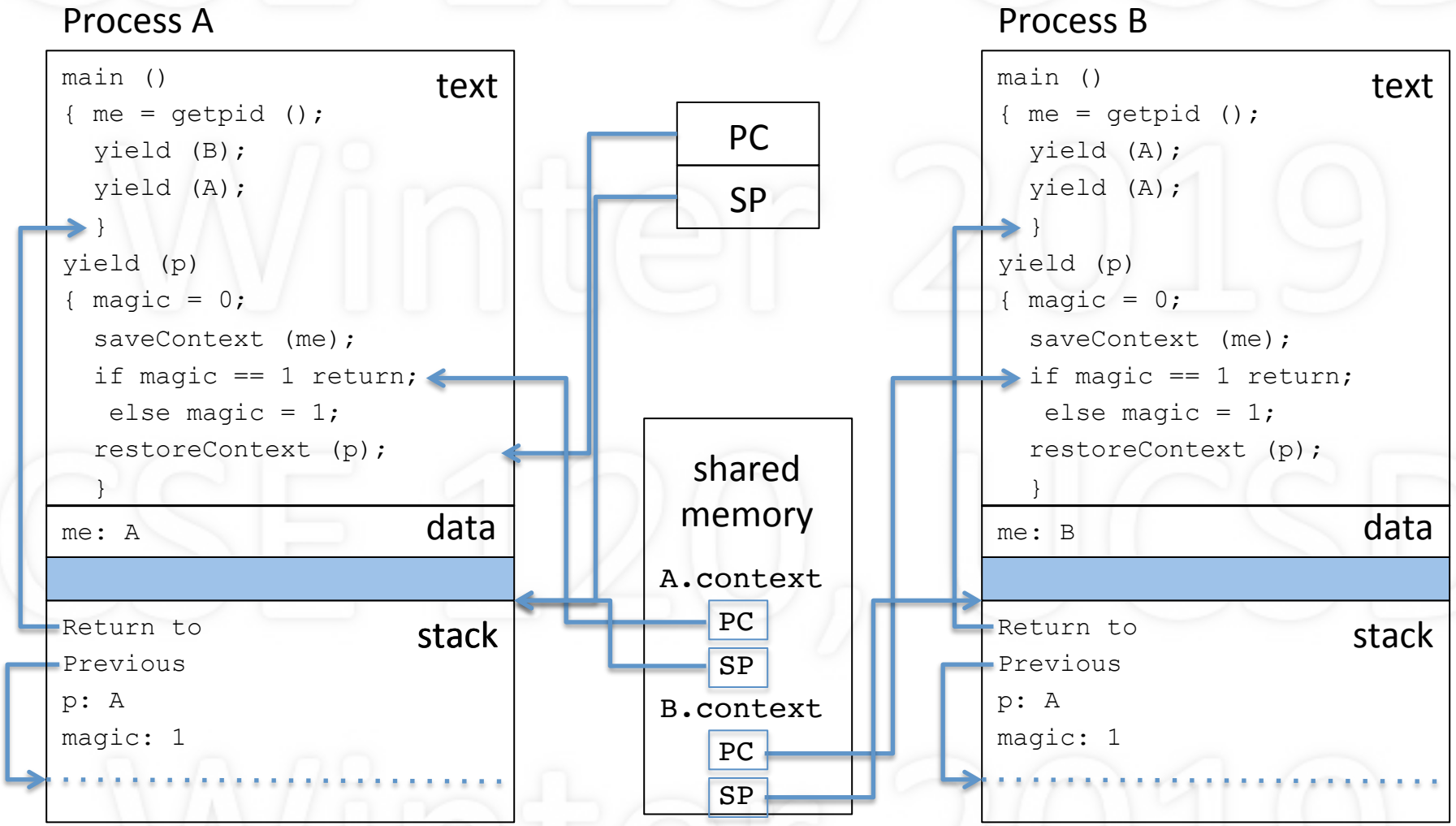
Note value of magic.



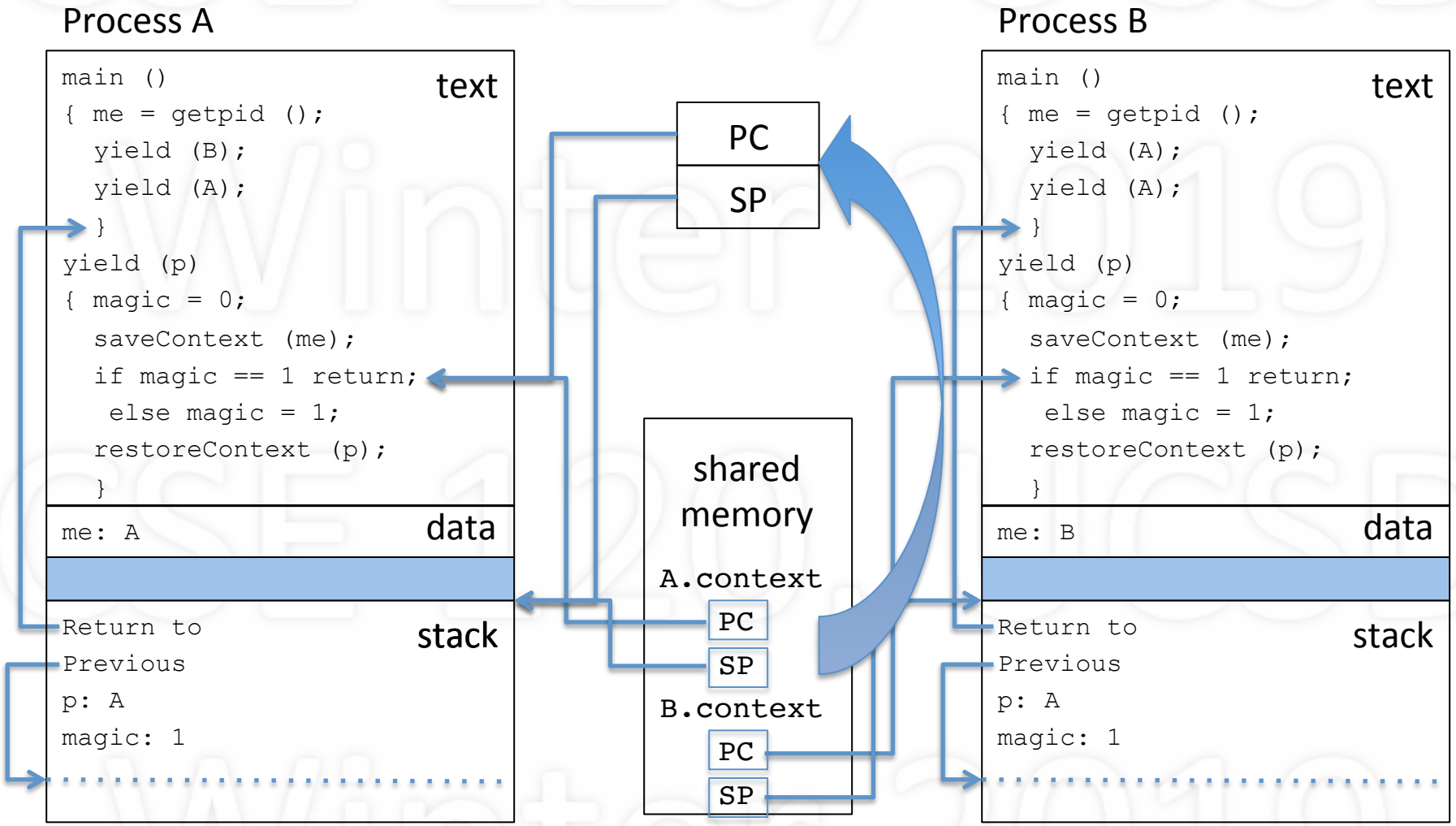
Set magic.



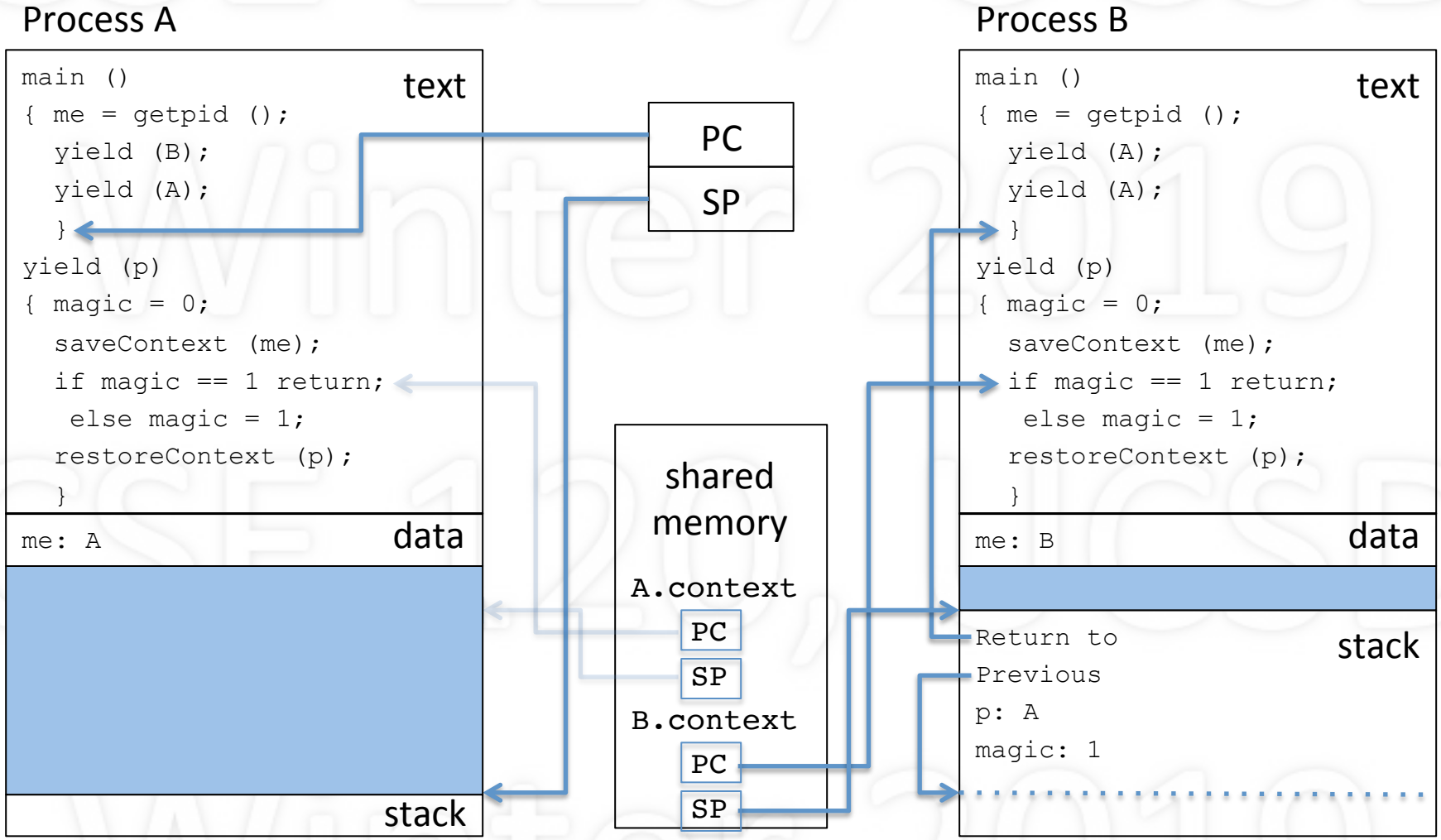
Restore the context of, A!



Note where A executes.



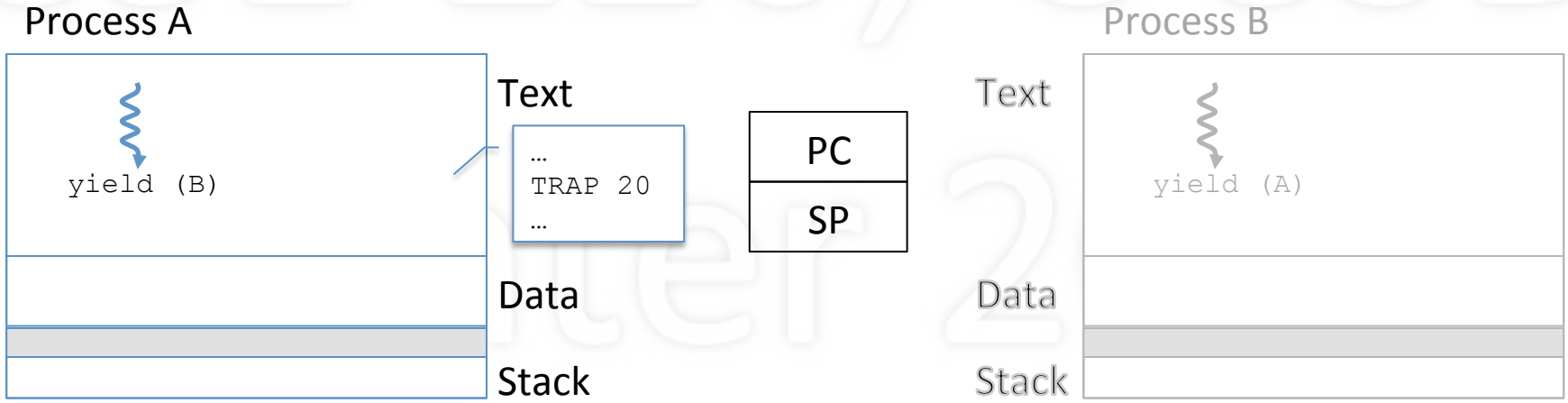
Return from yield.



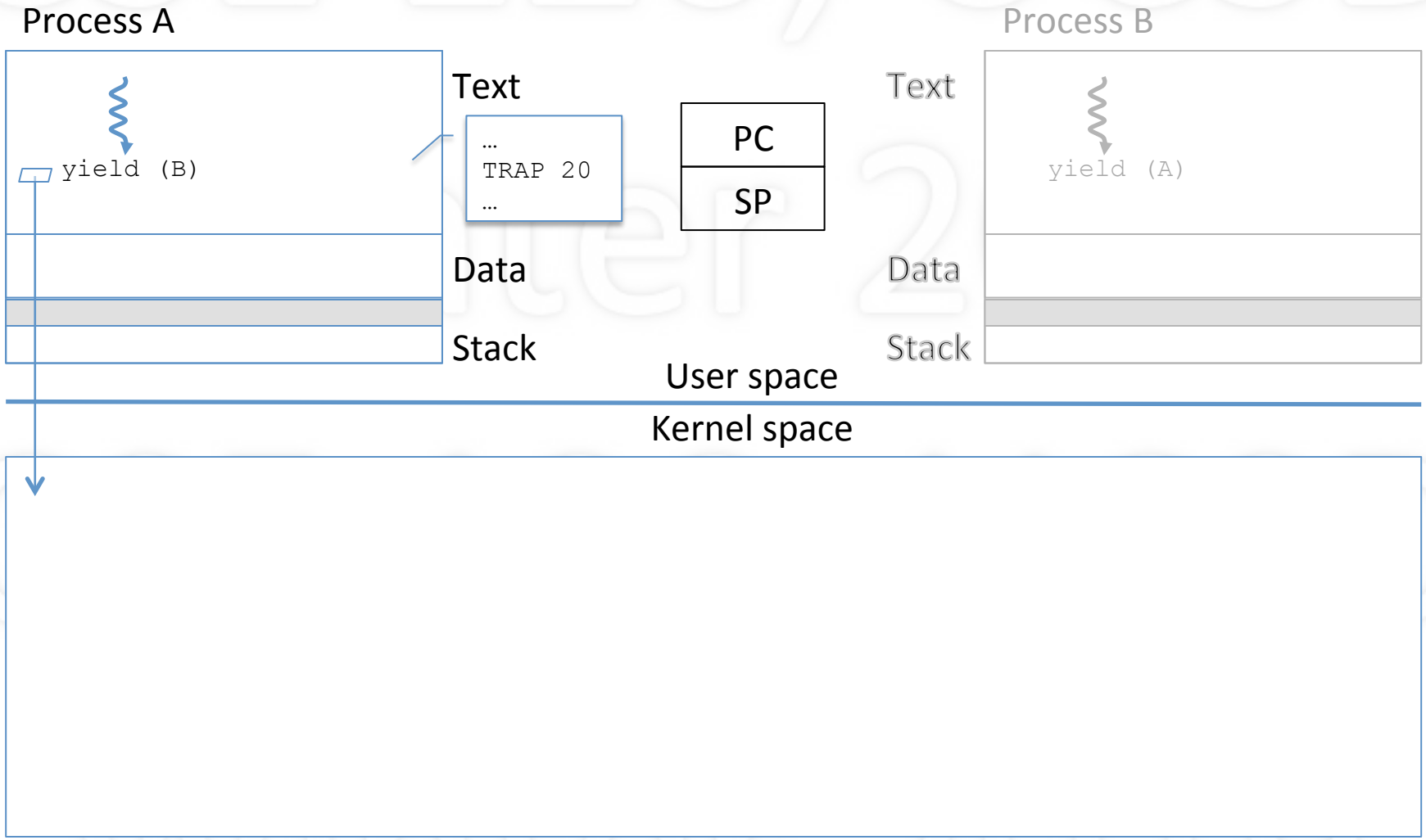
Yielding via the Kernel

- 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

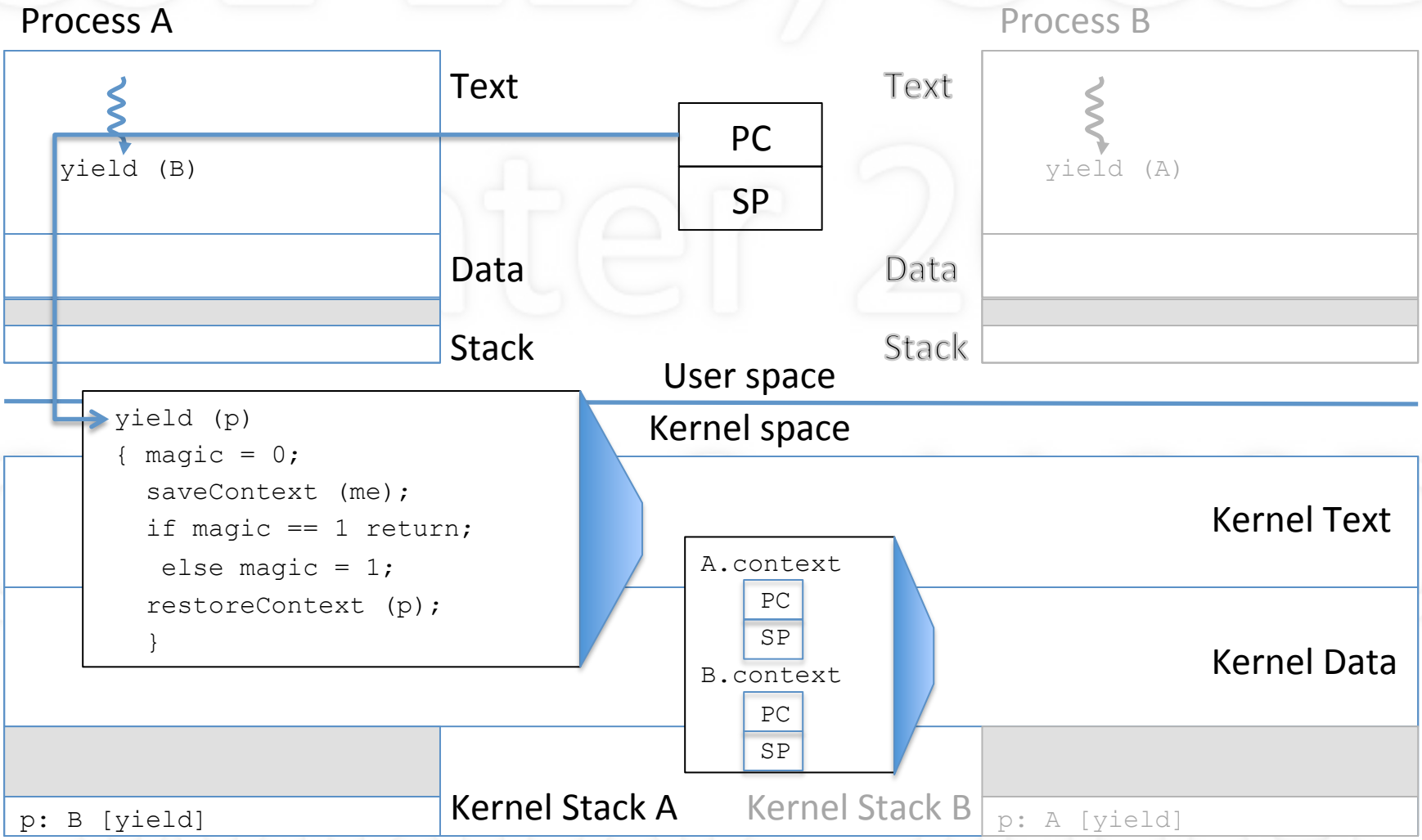
Yielding via the Kernel



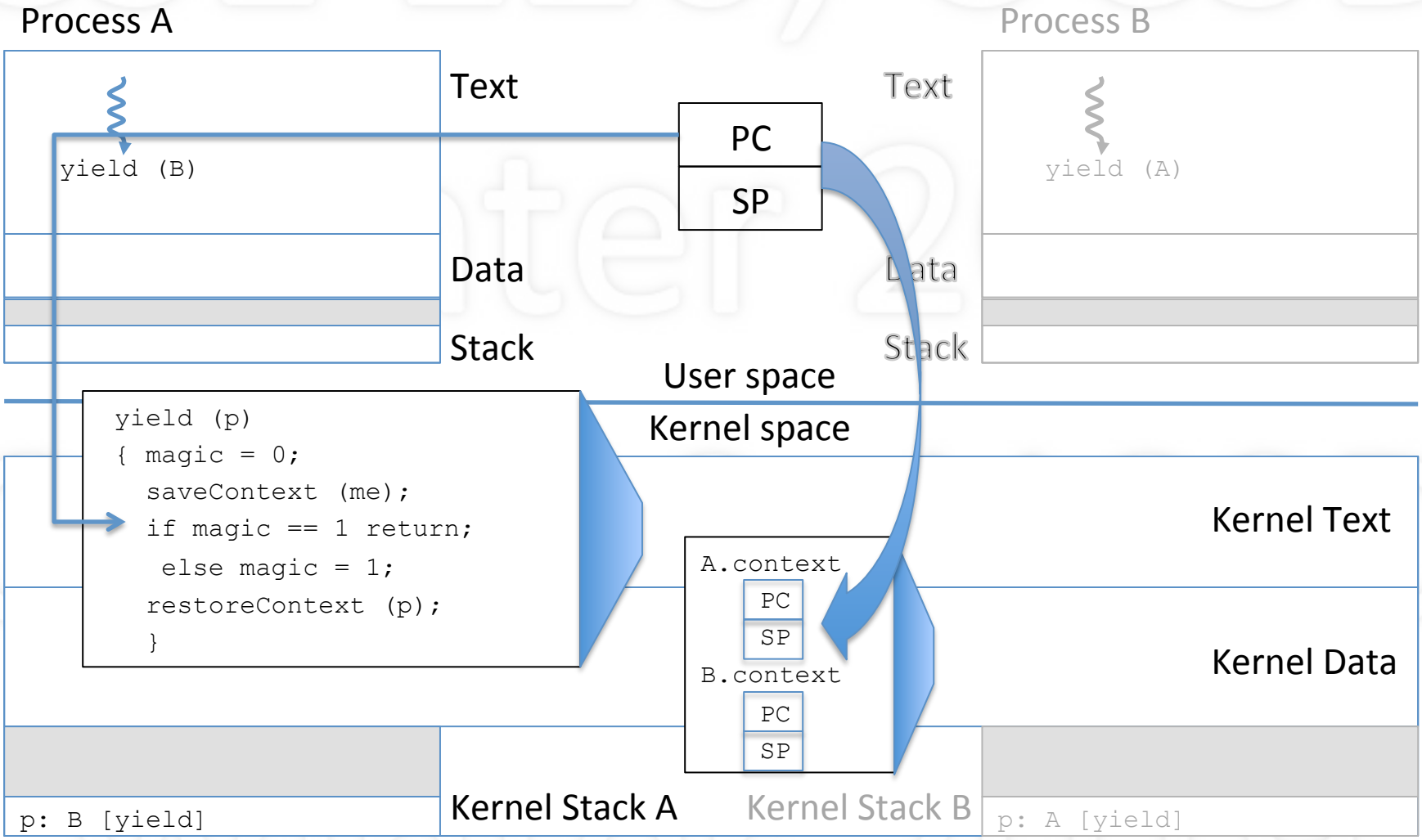
Trap Causes Kernel Entry



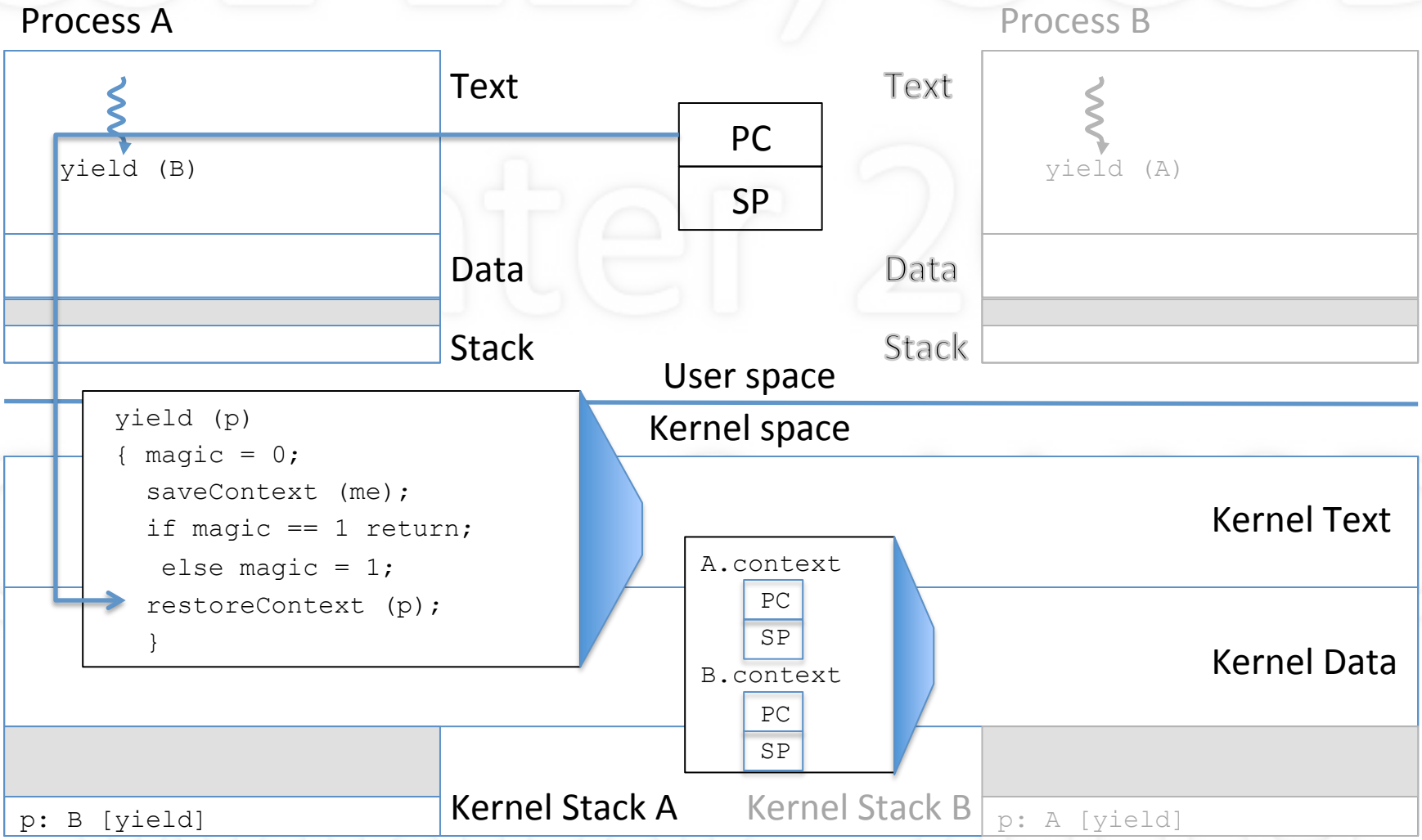
Routine for yield Executes



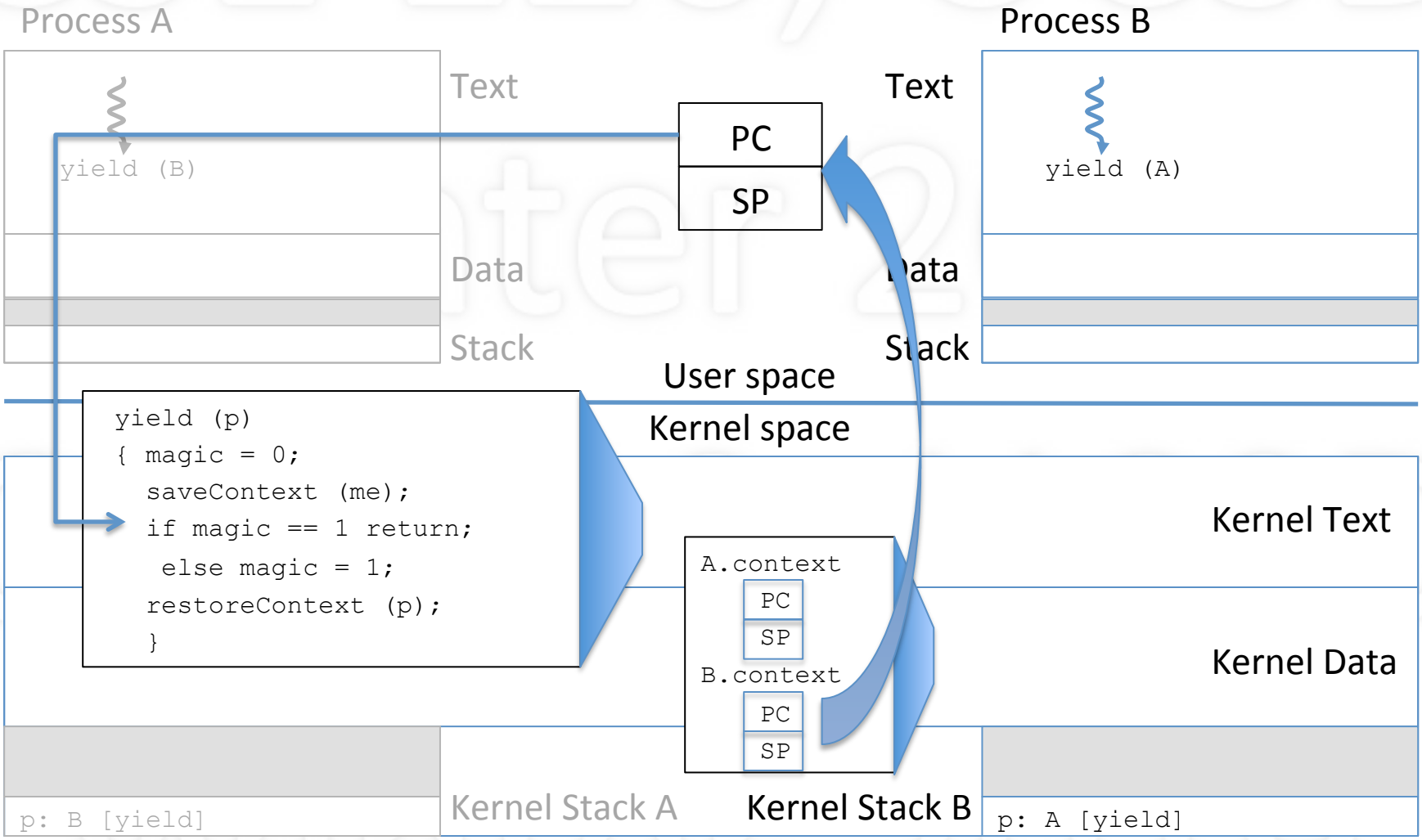
After Saving Context of A



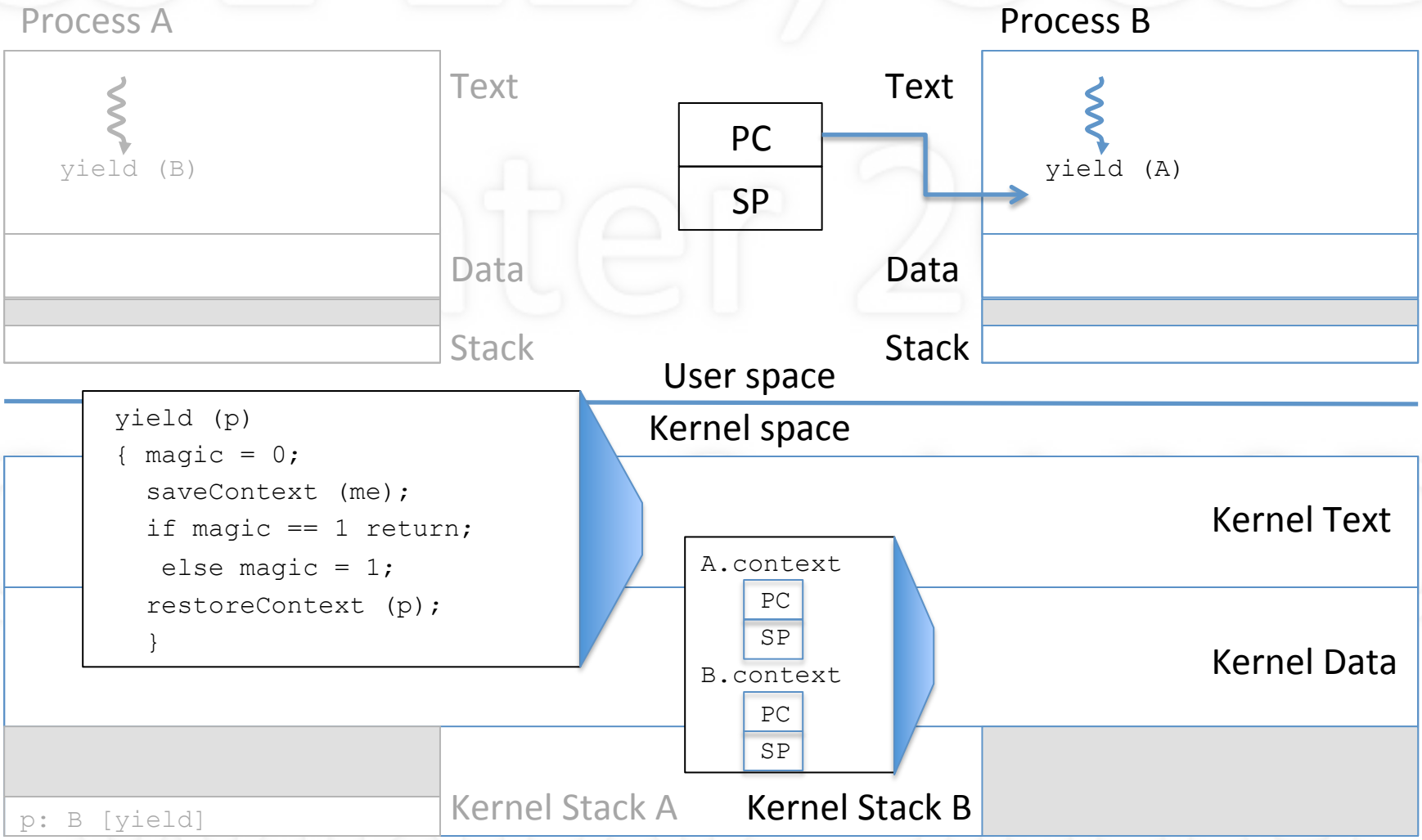
About to Restore Context of B



After Restoring Context of B



Return from yield in B



Summary

- Process
  - abstraction of a running program
- Multiprogamming
  - 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

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

Review & Research

- What is the most basic functionality that the kernel provides to users?
- Why is it difficult to run multiple programs on a machine with a single CPU and single (common) memory?**
- What is a process?
- How is a process different from a program?

R&R

- What basic resources does a process need?
- What is meant by the *context* of a process?
- What makes up the CPU context?
- What makes up the memory context?
- Looking up the word "context" in a dictionary, how does this general definition relate to how the word is used in operating systems?***

R&R

- What are the three sections that make up a process's memory structure?
- Why have three sections, as opposed to everything in one section?**
- How are global variables different from local variables in a C program?*
- In which section are global variables placed?
- Can they be placed in the other sections?**

R&R

- What is meant by *heap*?*
- Which section contains the heap?
- Can it be placed in the other sections?**
- What is an activation record?
- Which section contains activation records?
- Can they be placed in the other sections?**

R&R

- Which sections can grow and shrink?
- For each section that can grow and shrink, what event causes growth?* What event causes shrinking?**
- What is contained in the “SP” register?
- What functionality is enabled by the SP register?**
- Why does the SP register need to be part of the hardware?***

R&R

- What is contained in an activation record?
- Justify the need for each item of information: why does it need to be contained in the activation record?**
- Some machines provide an FP "frame pointer" register, in addition to an SP register: what is the purpose of the FP register, and why is it provided *in addition to* the SP register?***

R&R

- What is meant by: Users would like to run multiple programs “simultaneously”?**
- Why is “simultaneously” in quotes?***
- Why is running multiple processes difficult on a machine with a single CPU?**

R&R

- What is meant by *multiprogramming*?
- If a process needs a resource and it is busy, why do we say the process "voluntarily" gives up the CPU?**
- What does the yield routine do?
- What is meant by *context switching*?
- Why does yielding require context switching?*

R&R

- What are the basic steps done in context switching?*
- What must be the last instruction in context switching, and why?**
- In simple (i.e., 2-process) context switching, why is it necessary to switch text, data, and stack sections?*

R&R

- Which function is running during the switching of stacks?
- Just *before* switching stacks, what activation record (i.e., which function call does it correspond to) is at the top of the stack?**
- Just *after* switching stacks, what activation record (i.e., which function call does it correspond to) is at the top of the stack?***

R&R

- What is the role of the "magic" variable in yield (i.e., why is it needed)?**
- When running yield, where are the GP, SP, and PC saved?*
- When running yield, from where are the GP, SP, and PC restored?*
- How does yield know where to find the to-be-restored GP, SP, and PC?**

R&R

- In the Example program on p. 13, what does this program do?*
- How many processes are involved in this Example program?*
- On p. 14, why are the PC and SP pointing to the statements shown?
- What is contained in the shared memory?
- What is meant by *shared memory*?*

R&R

- Is the shared memory part of Process A or Process B?**
- Is “me” a global or local variable?
- Is “magic” a global or local variable and why?*
- Is a shared variable the same as a global variable?

R&R

- On p. 16, what caused the stack of process A to grow?*
- On p. 17, it says that saveContext and restoreContext are actually blocks of assembly language code: why?***
- What does saveContext do?*
- What does restoreContext do?*

R&R

- Since magic is initialized to 0, why does it make sense to even check whether it equals 1?**
- Will an optimizing compiler remove the if statement that checks whether magic equals 1, and why (or why not)?***

R&R

- On p. 22, what happened to the top activation record?**
- On p. 23, why is a new activation record created?**
- What is meant by “(residual)” next to “magic: 1”?**

R&R

- On p. 28, what is the value of magic for process A?
- How did it get set to this value?*

CSE 120, UCSD Winter 2019

R&R

- Why do we put the code for yield in the kernel?**
- Why do we put process contexts in the kernel?*
- Is the kernel a process, and why (or why not)? **
- How many memory sections does the kernel have, and what is their purpose?***

R&R

- What does the TRAP instruction do?**
- On p. 39, the TRAP instruction is shown as part of Process A: does this mean that the programmer actually typed in “TRAP 20” as part of the program; why or why not?***
- If the programmer did not type in “TRAP 20”, what generated that code?***

R&R

- On p. 40, what is meant by “Kernel Stack A”?*
- What is in Kernel Stack A, and why?**
- Why is the stack for Process A empty despite having called yield, whereas on p. 16, the stack for Process A grew after calling yield?**
- Does it matter which stack is used to store the activation record for yield, why or why not? ***

R&R

- On p. 44, what happened to Kernel Stack B?**
- How does the behavior of yield differ between slides 14-36 and slides 38-44?**
- Which method for implementing yield is better: that used in slides 14-36, or that used in slides 38-44?**
- Which method do you think is used in a real operating system, and why?***